Multi-Objective Memory Allocation-Based Code Optimizations for Real-Time Systems

Dissertation (monograph) approved by the

Doctoral Degree Committee of
Hamburg University of Technology

in pursuit of the academic degree of

Doktor der Naturwissenschaften (Dr. rer. nat.)

written by
Shashank Jadhav

from
Kolhapur, India

2026

Creative Commons License
This work is licensed under the Creative Commons License Attribution 4.0 (CC BY 4.0). This means that it can be duplicated and made publicly available, also commercially, as long as the author, the source of the text and above-mentioned license are referred to. The exact license text can be found under: https://creativecommons.org/licenses/by/4.0/legalcode.de.

DOI: 10.15480/882.18399
ORCID iD: 0000-0003-0980-880X

List of reviewers: Prof. Dr. Heiko Falk
Prof. Dr. Sibylle Schupp

Date of oral examination: 18 August 2026

Summary

Embedded systems are computer systems embedded into enclosing products dedicated to performing specific tasks consistently. They are often small, with limited memory and energy budgets. Embedded systems that must react within a fixed deadline are called real-time systems. Failure to meet such constraints might lead to disastrous consequences in real-life scenarios.

The thesis proposes various memory allocation-based multi-objective optimizations to optimize multiple objectives simultaneously. The thesis introduces a compiler-level static ScratchPad Memory (SPM) allocation-based multi-objective optimization that minimizes the program's Worst-Case Execution Time (WCET) and energy consumption and extends it for multi-task systems, which optimizes system schedulability as an objective along with WCET and energy consumption. The multi-objective optimization is solved using two metaheuristic algorithms—the Flower Pollination Algorithm (FPA) and the Strength Pareto Evolutionary Algorithm (SPEA). However, the metaheuristic algorithms are iterative and require multiple time-consuming WCET and energy analyses to bring the multi-objective optimization to fruition. Therefore, the thesis proposes three approaches to deal with the optimization time bottleneck: an approximation model-based approach, a Path-based Constraint Approach (PCA), and an Impact-based Constraint Approach (ICA).

The static SPM allocation has an SPM size constraint, which limits the number of basic blocks that can be allocated to the SPM statically and hinders the possible optimization potential. Therefore, the thesis proposes a compiler-level dynamic SPM allocation-based multi-objective optimization that helps us to deal with the SPM size constraint. Instead of statically allocating basic blocks once during the startup, the dynamic SPM allocation enables the dynamic allocation of functions and loops from one memory to another during runtime. However, copying of code dynamically incurs heavy WCET and energy costs and requires heavy code transformation, resulting in a code size increase. Therefore, the thesis proposes two approaches to deal with these issues: a Direct Memory Access (DMA)-aware approach and a compression-aware approach.

The thesis revolves around memory allocation techniques that can be exploited to optimize multiple objectives simultaneously. Each proposed optimization has its benefits and shortcomings. As the thesis progresses, the thesis proposes new approaches, introduces new concepts, builds upon previous optimizations, and tackles the previous optimization's shortcomings.

Contents

1. Introduction **1**
- 1.1. Contributions . . . 3
- 1.2. Structure . . . 5

I. Preliminaries **7**

2. Real-Time Embedded Systems **9**
- 2.1. Worst-Case Execution Time . . . 10
- 2.2. Energy Consumption . . . 13
- 2.3. Schedulability . . . 15
- 2.4. Embedded Architectures . . . 17
  - 2.4.1. Processors . . . 18
  - 2.4.2. Memory . . . 18
  - 2.4.3. Direct Memory Access (DMA) . . . 20
  - 2.4.4. ARM Cortex-M0 . . . 22

3. WCET-aware C Compiler (WCC) **23**
- 3.1. WCC Infrastructure . . . 24
  - 3.1.1. Memory Hierarchy . . . 26
  - 3.1.2. Flow Fact Specifications . . . 27
  - 3.1.3. Jump Correction . . . 29
- 3.2. WCET and Energy Consumption Analyzers . . . 33
- 3.3. Schedulability Analyzer . . . 35

4. Multi-Objective Fundamentals **37**
- 4.1. Single-Objective Optimization . . . 37
- 4.2. Integer Linear Programming (ILP) . . . 38
- 4.3. Multi-Objective Optimization . . . 39
  - 4.3.1. Problem Formulation . . . 40
  - 4.3.2. Pareto Optimality . . . 40
  - 4.3.3. Quality Indicators . . . 43
- 4.4. Metaheuristic Algorithms . . . 46
  - 4.4.1. Strength Pareto Evolutionary Algorithm (SPEA) . . . 48
  - 4.4.2. Flower Pollination Algorithm (FPA) . . . 49

II. Static Allocation **53**

5. Multi-Objective Static SPM Allocation **55**
5.1. Related Work . . . 57
5.2. Problem Formulation . . . 59
5.3. Multi-Objective Static SPM Allocation Optimization . . . 61
5.4. Evaluation . . . 63
5.4.1. Evaluation Setup . . . 64
5.4.2. Pareto Fronts . . . 66
5.4.3. Quality Indicators . . . 69
5.4.4. Compilation Times . . . 71
5.5. Conclusion . . . 72

6. Approximating WCET and Energy for Fast Multi-Objective Static SPM Allocation **73**
6.1. Related Work . . . 75
6.2. Problem Formulation . . . 76
6.3. Approximation Model . . . 77
6.3.1. Individuals for the Approximation Model . . . 78
6.3.2. Approximation Model for WCET . . . 81
6.3.3. Approximation Model for Energy Consumption . . . 84
6.3.4. Approximation Model's Limitations . . . 86
6.4. Multi-Objective Optimization . . . 86
6.5. Evaluation . . . 87
6.5.1. Evaluation Setup . . . 89
6.5.2. Compilation Times . . . 89
6.5.3. Pareto Fronts . . . 91
6.5.4. Approximation Errors . . . 94
6.5.5. Quality Indicators . . . 95
6.6. Conclusion . . . 96

7. Search Space Reduction of Multi-Objective Static SPM Allocation for Multi-Task Systems **97**
7.1. Related Work . . . 99
7.2. Problem Formulation . . . 100
7.3. Multi-Task Multi-Objective Static SPM Allocation Optimization 103
7.4. Path-based Constraint Approach (PCA) . . . 105
7.5. Impact-based Constraint Approach (ICA) . . . 107
7.6. Evaluation . . . 110
7.6.1. Evaluation Setup . . . 110
7.6.2. Compilation Times . . . 112
7.6.3. Pareto Fronts . . . 115
7.6.4. Quality Indicators . . . 118
7.7. Conclusion . . . 119

III. Dynamic Allocation 121

8. Multi-Objective Dynamic SPM Allocation 123
8.1. Related Work . . . 126
8.2. Problem Formulation . . . 128
8.2.1. Memory Objects . . . 128
8.2.2. Decision Variable Vector . . . 129
8.2.3. Address Objects . . . 132
8.2.4. Optimization Problem . . . 134
8.3. Dynamic SPM Allocation . . . 135
8.3.1. Contextual Overview . . . 135
8.3.2. Liveness Analysis . . . 136
8.3.3. Address Assignment Algorithm . . . 140
8.3.4. Code Generation . . . 142
8.3.5. Code Analyses . . . 144
8.4. Multi-Objective Dynamic SPM Allocation-based Optimization 145
8.5. Evaluation . . . 147
8.5.1. Evaluation Setup . . . 147
8.5.2. Pareto Fronts . . . 148
8.5.3. Quality Indicators . . . 153
8.5.4. Compilation Times . . . 154
8.6. Conclusion . . . 155

9. DMA-aware Multi-Objective Dynamic SPM Allocation 157
9.1. Related Work . . . 161
9.2. Fundamentals of Dynamic SPM Allocation using DMA . . . 162
9.2.1. Problem Formulation . . . 162
9.2.2. Contextual Overview . . . 165
9.2.3. Code Generation and Analyses . . . 167
9.3. DMA Call Placement Optimization . . . 169
9.3.1. Contextual Overview . . . 169
9.3.2. Cost Calculation . . . 173
9.3.3. Optimization Problem . . . 183
9.4. DMA-aware Multi-Objective Dynamic SPM Allocation . . . 186
9.5. Evaluation . . . 187
9.5.1. Evaluation Setup . . . 188
9.5.2. DCPO Evaluation . . . 189
9.5.3. Optimizations Comparison with 60% Relative SPM Size and DCPO–FPA . . . 192
9.5.4. Optimizations Comparison with 60% Relative SPM Size and DCPO–ILP . . . 200
9.5.5. Optimizations Comparison with 20%, 40%, and 80% Relative SPM Size . . . 204
9.6. Conclusion . . . 208

10. Compression-aware Multi-Objective Dynamic SPM Allocation 211
10.1. Related Work . . . 214
10.2. Dynamic SPM Allocation . . . 215
10.2.1. Problem Formulation . . . 215
10.2.2. Contextual Overview . . . 218
10.3. Dynamic Compression . . . 221
10.3.1. Contextual Overview . . . 221
10.3.2. Resolving Function Call Dependencies . . . 222
10.3.3. Function Compression . . . 225
10.3.4. Code Generation and Analyses . . . 226
10.4. Compression-aware Multi-Objective Dynamic SPM Allocation 229
10.5. Evaluation . . . 230
10.5.1. Evaluation Setup . . . 231
10.5.2. Pareto Fronts . . . 232
10.5.3. Code Size Comparison . . . 235
10.5.4. Quality Indicators . . . 237
10.5.5. Compilation Times . . . 239
10.6. Conclusion . . . 240

IV. Conclusion 241

11. Conclusion and Future Work 243
11.1. Summary . . . 243
11.2. Future Work . . . 245

Bibliography 251

Acronyms and Notational Conventions 265
Acronyms . . . 265
Acronyms for Optimizations . . . 266
Symbols Used for Optimizations . . . 266
Metaheuristic Algorithms . . . 267
Static Allocation . . . 267
Dynamic Allocation . . . 268
DMA Call Placement Optimization . . . 269
Compression . . . 269

List of Figures 271

List of Tables 277

Listings 281

Appendices 283

Appendix A. Multi-Objective Static SPM Allocation **285**

Appendix B. Approximating WCET and Energy for Fast Multi-Objective Static SPM Allocation **289**

Appendix C. Search Space Reduction of Multi-Objective Static SPM Allocation for Multi-Task Systems **295**

Appendix D. Multi-Objective Dynamic SPM Allocation **305**

Appendix E. DMA-aware Multi-Objective Dynamic SPM Allocation **309**

Appendix F. Pipeline-aware DMA Cost Model **329**
F.1. Basic Block-level Net Execution Time 330
F.2. Basic Block-level DMA-aware Total Execution Time 332
F.3. Basic Block-level Parallel Execution Time 335
F.4. Considerations for Loops and Checks for the DMA Transfer Completions 335
F.5. Basic Block-level Placement Cost in Arbitration Mode 338
F.6. Total Execution Time Cost in Arbitration Mode 339

Appendix G. Compression-aware Multi-Objective Dynamic SPM Allocation **341**

1

Introduction

We are surrounded by computers that help us with our daily lives. Laptops, desktops, and smartphones are the obvious computers we use for work and leisure. However, small, tiny computers called embedded systems always work behind the scenes to improve our quality of life. To quickly heat food, we use microwave ovens; to regulate the room temperature, we use air conditioners; to track fitness activity, we use smartwatches. Embedded systems not only help improve the quality of life but are also used for safety and security purposes, such as car braking systems, airbags, flight controls, and fire alarms.

Embedded systems are hardware and software combined to perform specific tasks repeatedly without fail. Real-time systems are embedded systems that must perform a task within a certain deadline. A soft real-time system may tolerate a deadline miss, causing harmless consequences. For example, a printer is a soft real-time system; it should ideally operate on time; however, if a printer is delayed, it causes slight inconvenience, not a disaster. A soft real-time system missing a deadline might degrade the user experience and require the user to replace or repair the system.

However, a hard real-time system must not miss the deadline. A hard real-time system missing a deadline might have disastrous consequences. For example, a pacemaker helps keep the heartbeat at a normal rate and rhythm—failure of a pacemaker can have dire consequences. We need to consider the Worst-Case Execution Time (WCET) of a hard real-time system to address deadline misses in such systems. WCET is the worst possible execution time of a program, independent of its input data.

One of the key properties of embedded systems is mobility. Early computers were the size of a room. However, now we have more computing power on smartwatches than in those early computers. Mobile embedded systems imply a lack of continuous power supply. Therefore, we need such embedded systems to be energy-efficient.

Over the years, there have been remarkable developments in memory technologies, characterized by a decrease in physical size and an increase in capacity. However, embedded systems are getting complex and physically small, and requiring them to perform complex tasks can negatively impact the code size of the programs. Therefore, we need to consider the code size of a program while designing embedded systems.

WCET, energy consumption, and code size are some of the properties that a system designer might need to consider. However, simultaneously optimizing

for such properties is not straightforward, i.e., a system designer cannot know beforehand the changes to the program to obtain the optimal results. Currently, the design process for embedded systems is riddled with human interventions. A model-based approach is often used to design embedded systems graphically using tools like MATLAB. These tools generate C code from the graphical specifications and compile the generated C code. However, these general compilers do not have the timing and energy consumption notion of the underlying architecture. To optimize compiled code, the system designer can manually test, profile, and analyze the code to determine the timing and energy usage of the generated code. If the generated code does not satisfy the system constraints, the designer needs to make manual changes either in model specifications, in generated C code, or in compiler command-line options, and hope that with these new changes, the resulting system design satisfies all the constraints.

A system designer can make changes in the hardware specifications of the system to meet deadlines, improve energy consumption, or fit the program within the memory. However, hardware changes are costly and less flexible compared to software changes. Moreover, the more human interventions we have in the design process, the more faults we might introduce in the design. Therefore, we can use compilers as a tool to analyze and optimize program properties like WCET, energy consumption, and code size without human intervention.

In this thesis, we use the WCET-aware C Compiler (WCC) [FL10] to perform compiler-level analyses and optimizations. The WCC is capable of performing WCET, energy consumption, schedulability, and code size analyses without any human intervention. This thesis exploits the analysis capacity of WCC to propose and perform various compiler-level optimizations.

A single-objective optimization, as the name suggests, aims to optimize (minimize or maximize) a single objective function. They are characterized by their constraint conditions and weighted-summed or standalone optimization objectives. If the solution exists, the single-objective optimization has a unique optimal solution. On the contrary, multi-objective optimization has to juggle multiple objectives simultaneously that often contradict each other. For example, optimizing for the WCET of a program can negatively impact energy consumption and vice versa. Moreover, a multi-objective optimization might have multiple solutions showing trade-offs between the objectives. In this thesis, we propose various multi-objective optimizations and a single-objective optimization that aims to optimize different objectives at compile-time.

Single-objective optimization can be formulated as an Integer Linear Programming (ILP) problem and solved to find optimal solutions deterministically. However, we must be able to formulate all the constraint conditions and objective functions as linear functions to use ILP-based techniques. In this thesis, we use ILP-based techniques to solve single-objective optimizations. However, ILP-based approaches encounter limitations while solving multi-objective op-

timizations. Metaheuristic algorithms are widely used to solve multi-objective optimizations by iteratively updating solution sets and exploring the problem's solution search space.

Contrary to ILPs, metaheuristic algorithms are capable of juggling multiple objectives simultaneously. However, they do not guarantee optimality. In this thesis, we use metaheuristic algorithms to solve multi-objective optimization problems. Due to the iterative nature of metaheuristic algorithms, they can be time-consuming. Therefore, we propose novel methods to speed up metaheuristic algorithms at compile time.

Memories are a vital part of an embedded system. We can use differences in memory characteristics to optimize a program during compilation. In this thesis, we propose various static and dynamic memory allocation-based optimizations that aim to optimize multiple objectives by placing program fragments in different memories. Static memory allocation is a compiler-level optimization that allocates program fragments from one memory to another statically at compile time; dynamic memory allocation allocates program fragments from one memory to another dynamically during runtime.

We demonstrate the efficacy of memory allocation techniques by using the Flash memory and the ScratchPad Memory (SPM). SPMs are fast and energy-efficient memories, but they are small compared to the Flash. We exploit the benefits and shortcomings of SPMs to propose novel static and dynamic memory allocation-based multi-objective optimizations in the upcoming chapters.

1.1 Contributions

In this thesis, we focus on SPM allocation-based multi-objective optimizations for real-time embedded systems, aiming to minimize multiple objectives simultaneously. We formulate compiler-level static and dynamic SPM allocation-based optimizations as multi-objective optimizations to find trade-offs between WCET, energy consumption, and code size of a program. We divided the thesis into four distinct parts. The first part of the thesis deals with the preliminaries and provides comprehensive background information required for the upcoming two parts. The second part of the thesis deals with static memory allocation, and the following are the major contributions of the second part:

- We formulate a compiler-level multi-objective static SPM allocation optimization problem to optimize the program's WCET and energy consumption simultaneously. We solve this optimization problem using two metaheuristic algorithms—the Flower Pollination Algorithm (FPA) and the Strength Pareto Evolutionary Algorithm (SPEA).

- We extend the multi-objective static SPM allocation optimization to handle multi-task systems that aim to simultaneously minimize three objectives—WCET, energy consumption, and schedulability.

- We also show that solving multi-objective static SPM allocation iteratively using metaheuristic algorithms is time-consuming due to the requirement of multiple WCET and energy consumption objective analyses.

- We propose three approaches to deal with the compilation time bottleneck while solving multi-objective static SPM allocation optimization:
  - We propose an approximation model that approximates WCET and energy consumption of a statically allocated code. We use the proposed approximation model instead of time-consuming analyzers to evaluate the individual solutions while running metaheuristic algorithms.
  - We propose a Path-based Constraint Approach (PCA) that uses the program's worst-case execution and energy path information to constrain the solution space. By constraining the solution space, we aimed to reduce the required number of iterations for completing the metaheuristic algorithm runs.
  - We propose an Impact-based Constraint Approach (ICA) that uses a so-called *Impact* metric to constrain the solution space. The *Impact* metric depends on the program's worst-case execution and energy information.

- The approximation model-based approach focuses on WCET and energy consumption as objectives. The results show that the proposed approximation model drastically reduces the optimization time with only minimal approximation errors while maintaining the solution quality.

- Both PCA and ICA approaches aim to simultaneously minimize WCET, energy consumption, and schedulability for multi-task systems. PCA and ICA successfully constrain the search space and reduce the required optimization time. However, ICA provided better Pareto optimal solutions.

- To demonstrate the proposed approaches, we evaluated the approximation model, PCA, and ICA techniques using the ARM Cortex-M0 fork of the WCC compiler.

As static SPM allocation is constrained by small SPM size constraints, we focus the third part of the thesis on dynamic SPM allocation techniques. The following are the major contributions of the third part of the thesis:

- We formulate a compiler-level multi-objective dynamic SPM allocation optimization that does not have an SPM size constraint and aims to minimize multiple objectives simultaneously. The dynamic allocation enables us to dynamically allocate functions and loops from Flash to SPM during runtime.
- Dynamic allocation incurs heavy WCET and energy consumption costs. Dynamic allocation also requires heavy code transformation, increasing the code size. Therefore, we further propose two novel approaches to deal with these issues:
  - To tackle the heavy WCET and energy costs, we integrate the Direct Memory Access (DMA) model with dynamic SPM allocation at the compiler level to use the DMA controller instead of the CPU for efficient dynamic allocation of objects during runtime.
  - To tackle the increase in code size issue, we propose a compression-aware multi-objective dynamic SPM allocation to minimize three objectives—WCET, energy consumption, and code size.
- We propose a compiler-level DMA model, the accompanying compiler-level DMA analyzer, and a single-objective DMA Call Placement Optimization (DCPO) to find optimal places within the program to place DMA transfer calls for dynamic allocation. We solve and evaluate the DCPO problem using three methods—ILPs, FPA, and SPEA.
- For compression-aware multi-objective dynamic SPM allocation optimization, we compress dynamic allocation objects at compile time and store them in a data buffer. The compressed objects are decompressed dynamically during runtime in SPM. The metaheuristic algorithm solves the multi-objective optimization iteratively to optimize all the objectives simultaneously.
- To demonstrate the proposed approaches, we evaluated all the dynamic allocation-based approaches using the ARM Cortex-M0 fork of WCC.

The final fourth part of the thesis discusses the conclusions from this thesis and the possible future works.

1.2 Structure

This thesis is structured as follows:

- Chapter 2 defines embedded systems, real-time systems, and their fundamental properties. It also provides background information about embedded architectures and their components.

- Chapter 3 presents the WCC compiler framework used to implement and evaluate the optimizations proposed in the thesis. It also provides brief information concerning the infrastructure for WCET, energy consumption, and schedulability analyses within WCC.
- Chapter 4 provides background information regarding single- and multi-objective information. It also presents brief information on the ILPs and metaheuristic algorithms (FPA and SPEA).
- Chapter 5 proposes the multi-objective static SPM allocation optimization for single-task real-time systems to optimize the program's WCET and energy consumption simultaneously.
- Chapter 6 presents a novel approximation model to approximate WCET and energy objectives for multi-objective static SPM allocation for single-task real-time systems.
- Chapter 7 expands the multi-objective static SPM allocation optimization to handle multi-task systems and aims to optimize three objectives—WCET, energy consumption, and schedulability. It also proposes PCA and ICA approaches to constrain the search space of the optimization problem and deal with the compilation time bottleneck.
- Chapter 8 proposes a novel multi-objective dynamic SPM allocation optimization for single-task real-time systems to optimize the program's WCET and energy consumption. It dynamically allocates functions and loops to SPM during runtime.
- Chapter 9 expands the multi-objective dynamic SPM allocation and the WCC compiler to use the DMA controller instead of the CPU during runtime dynamic allocation. It proposes a DMA cost model to calculate the extra execution time costs incurred during DMA transfers. It also proposes a novel single-objective DMA Call Placement Optimization (DCPO) to find optimal places within the code for DMA transfer calls needed for dynamic allocation.
- Chapter 10 proposes a novel compression-aware multi-objective dynamic SPM allocation optimization that aims to compress functions and loops at compile time and decompress them into SPM dynamically during runtime to optimize three objectives simultaneously—WCET, energy consumption, and code size.
- Chapter 11 closes the thesis, providing a conclusion and a discussion on the possible future works.

Part I

Preliminaries

2

Real-Time Embedded Systems

Embedded systems have become part of everyday life. Charles Stark Draper is credited with developing the first so-called modern embedded system for the Apollo Guidance Computers in the 1960s. Nowadays, embedded systems work tirelessly in the background so well that we might fail to recognize their presence or contribution to our lives.

Embedded systems are a combination of computer processors, memories, input/output peripherals, and software designed independently or as part of a larger system to perform specific tasks, for example, drones, cameras, air conditioners, etc. Embedded systems function as part of a larger system or a complete device—giving them the name embedded. Marwedel [Mar11] used the following definition for embedded systems.

Definition 2.1 (Embedded Systems)
Embedded systems are information processing systems embedded into enclosing products.

The following additional points can characterize an embedded system:

- Need to perform specific tasks consistently
- Need to be reliable to maintain such consistency
- Often need to perform tasks under timing constraints to keep the functionality of the larger system correct
- Often need to perform tasks on a limited energy budget
- Often has limited memory, low cost, and lightweight

Software development for such embedded systems differs from classical software development. Embedded software needs to target underlying embedded hardware—posing memory constraints on the software. The nature of embedded applications can pose constraints on software's non-functional properties, such as execution time and energy consumption. Embedded systems that have to comply with timing deadlines are called *Real-Time Embedded Systems*. Manacher [Man67] used the following definition for real-time systems.

Definition 2.2 (Real-Time System)
A computing system that has to finish its calculation and return the corresponding results within a given amount of time is called a real-time system.

Kopetz [Kop11] used the following definition for real-time embedded systems.

Definition 2.3 (Real-Time System)
A real-time system is an embedded system where the correctness of the system behavior depends not only on the logical results of the computations but also on the physical time when these results are produced.

Real-time systems can be categorized depending on the nature of the timing constraint. Real-time systems of a statistical nature that require a response in a reasonable fashion and can tolerate small variations in response are called *soft real-time systems*. Systems with rigid timing bounds and beyond which they will not be able to make use of processed data are called *hard real-time systems* [Man67]. Kopetz [Kop11] defined a *hard* or *safety-critical real-time system* as a system with at least one hard deadline; *soft real-time systems* have no hard deadlines. Missing occasional soft deadlines is permissible, whereas failing to comply with hard deadlines can have disastrous consequences. For example, multimedia systems are soft real-time systems—no harm is caused if the multimedia system misses a deadline. On the other hand, railway signaling systems are hard real-time systems—timely railway signaling to control the movement of railway traffic is needed to avoid catastrophic harm.

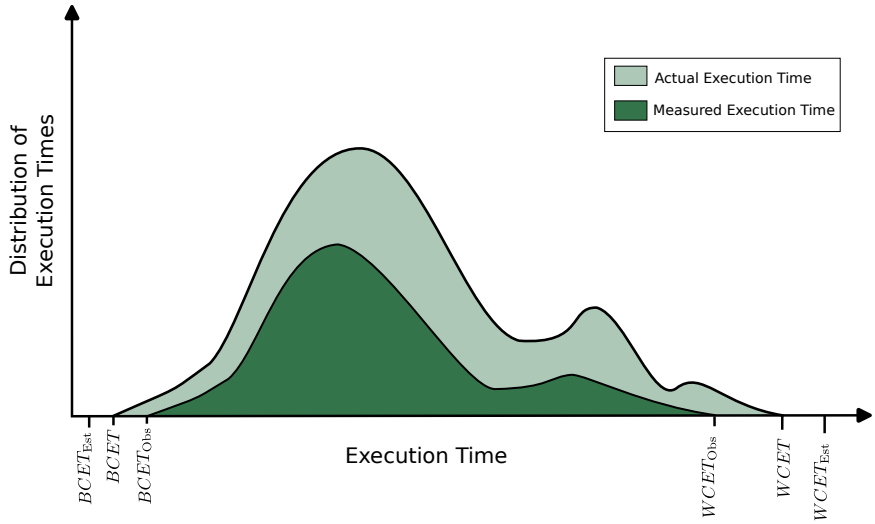
This chapter is organized as follows: Sections 2.1, 2.2, and 2.3 discuss execution times, energy consumption, and schedulability properties of real-time embedded systems. Section 2.4 provides information about embedded architectures, memories, Direct Memory Access (DMA), and the embedded processor considered in this thesis.

2.1 Worst-Case Execution Time

The execution time of a given program is the time spent by the system executing that task. Timing analysis is used to inspect the program and obtain the program's execution time. Depending on the type of analysis, we can obtain the best-, the average-, or the worst-case execution time of the program. Worst-Case Execution Time (WCET) plays a crucial role in developing real-time systems. Ermedahl [Erm03] used the following definition for WCET.

Definition 2.4 (WCET)
WCET is the longest execution time of a program that could ever be observed when the program is run on its target hardware.

Figure 2.1.: Exemplary distribution of all possible execution times of a program and distribution of measured execution times adapted from Ermedahl [Erm03] and Wilhelm et al. [WEE+08].



Therefore, WCET is used to determine if a real-time program fulfills its deadlines. For a multi-task system, WCET describes the longest possible execution time without interference from other tasks. The program's execution time can vary due to its input data and the state of the processor on which it is executed. Marwedel [Mar11] provided the following tighter definition for the program's WCET.

Definition 2.5 (WCET)
The WCET of a program is the largest execution time of the program for any input and any initial execution state.

Determining the WCET of a program becomes unattainable due to these "any input" and "any initial execution state" conditions. Alan Turing [T+36] showed that we cannot determine whether an arbitrary program for any input will terminate or run forever, i.e., no general algorithm exists that solves the halting problem for all possible inputs. Calculating the WCET of a program requires us to solve the halting problem or to know the worst-case input of the program. However, the halting problem is unsolvable, and determining the worst-case input is impossible, as the state space can be massive and impossible to explore in a limited time.

Figure 2.1 provides an example of the distribution of execution times for a set of given inputs (indicated in the figure with dark green color) and all possible inputs (indicated in the figure with light green color), respectively. For a set of given inputs, we can measure the execution time of any program. However, measuring or analyzing any program for all possible inputs is impossible. Moreover, knowing if the worst-case input exists in the set of all given inputs is also impossible.

Remark 2.1
The distribution presented in Figure 2.1 shows a hypothetical example for illustration, as it is impossible to get such a distribution for real-life programs.

Timing analysis is done using three main techniques: measurement-based analysis, static analysis, and probabilistic measurement-based analysis. Wilhelm et al. [WEE+08] and Nélis et al. [NYP22] provide overviews of the different timing analysis methodologies for WCET analysis.

- **Measurement-based Analysis:** The measurement-based technique uses execution time measurements for some given inputs from the real hardware or simulators to determine the program's execution times. This technique provides observed Best-Case Execution Time ($BCET_{Obs}$) and Observed WCET ($WCET_{Obs}$), which are optimistic estimates of the true BCET and WCET, i.e., $BCET < BCET_{Obs}$ and $WCET_{Obs} < WCET$. Therefore, measurement-based analysis is not safe for developing real-time systems. However, they can be used to develop embedded systems where average-case behavioral analysis is necessary, i.e., measurement-based techniques are best to determine Average-Case Execution Time (ACET).

- **Static Analysis:** Static analysis techniques estimate safe upper bounds for WCET ($WCET_{Est}$) and safe lower bounds for BCET ($BCET_{Est}$) using a mathematical model of the underlying architecture, i.e., $WCET < WCET_{Est}$ and $BCET_{Est} < BCET$. The tightness of these bounds depends on the predictability of the architecture and the availability of quality information regarding the hardware. The tighter the bounds, the better the estimation. Therefore, the WCET and BCET estimations should be as close to the actual WCET and BCET as possible, i.e., $WCET_{Est} - WCET \rightarrow 0$ and $BCET - BCET_{Est} \rightarrow 0$. Static analysis is used for real-time systems as it guarantees that $WCET_{Est}$ and $BCET_{Est}$ are safe.

  Hardware- and software-determinism is crucial while statically estimating a program's WCET. We must consider memory access patterns, cache behavior in the worst case, external I/O hardware delays, pipeline stalls, etc. Moreover, the program's control flow needs to be analyzable and predictable. For example, we need well-defined branches, bounded recursion, loop bounds information, etc., while performing static WCET analysis. WCET analysis becomes more complicated with increasing architectural complexities like shared buses, shared memories, multi-core systems, etc.

- **Probabilistic Measurement-based Analysis:** Static analysis needs profound knowledge of the underlying architecture to provide tighter $WCET_{Est}$ and $BCET_{Est}$. In the absence of such knowledge, probabilistic measurement-based techniques can determine the upper bound of probabilistic WCET distributions from measurements. Extreme value theory

is used to generate a statistical model of execution time distributions and derive conclusions on the probability of execution times greater than those observed during measurements. However, as the name suggests, the probabilistic analysis does not provide safe guarantees.

As the static analysis technique provides safe $WCET_{Est}$ and does not require any measurements, it is ideal for performing compiler-level WCET analysis without human intervention. A static analysis tool called `aiT` [Abs24] is used in this thesis to perform compiler-level WCET analysis[1].

2.2 Energy Consumption

Many modern real-time systems are battery-operated, for example, pacemakers, drones, etc. They must be energy-efficient, as energy consumption plays a critical role in prolonging the functioning of these devices. The focus lies on energy-intensive components of the real-time system to estimate the energy consumption of a program, namely, the processor and memories. Guo et al. [GCZY21] surveyed different energy consumption measurement techniques used for embedded systems.

Simunic et al. [ŠBDM99] proposed an energy model based on component datasheets. Each considered component was assumed in either an active or idle state, and by providing them with energy values in both states, they built an energy model. They extended a cycle-accurate instruction-level simulator with the proposed energy model to simulate the energy consumption in embedded systems. Such coarse-grained models can be imprecise, and the availability of detailed information regarding the hardware is also ambitious.

Tiwari et al. [TMW94] presented the first instruction-based energy model that considered the base costs for instructions and inter-instruction costs. The processor executes a long sequence of the same instruction to measure the average processor current, and the base cost for each instruction is calculated. The inter-instruction cost was determined by executing long sequences of pairs of different instructions and using the base cost information to capture the energy costs during the transition from one instruction to the other. The total energy consumption of a program q is calculated using:

$$E_q = \sum_i B_i N_i + \sum_{i,j} I_{(i,j)} N_{(i,j)} + \sum_k E_k \tag{2.1}$$

where B_i is the base cost of instruction type i, $I_{(i,j)}$ is the inter-instruction cost between types i and j, N_i is the number of instructions of type i, $N_{(i,j)}$ is the number of instruction pair type (i,j), and E_k is the energy consumed by runtime instruction effects k, i.e., cache misses, pipeline stalls, etc.

Steinke et al. [SKWM01] extended this work by considering the Hamming distance between the current and the succeeding instruction words. They

[1] A detailed explanation of `aiT` is provided in Chapter 3.

proposed a measurement-based memory energy model that used the Average-Case Execution Counts (ACEC) of a program to estimate average-case energy consumption. The total energy consumption of a program q according to this model is:

$$E_q = E_{pi} + E_{pd} + E_{mi} + E_{md} \tag{2.2}$$

where E_{pi} and E_{pd} are the instruction- and the data-dependent energy costs for the processor, and E_{mi} and E_{md} are the energy costs for the instruction and data memories.

Konstantakos et al. [KCNL08] proposed an energy model that depends on the number of cycles taken by executed instructions, the activity of the A/D converter, and the number of memory accesses. Profiling was used to analyze the program, and a measurement setup was used to model the energy behavior of each component during active and standby states. The total energy consumption of program q according to this model is:

$$E_q = E_{mc} + E_r + E_{ad} \tag{2.3}$$

where E_{mc}, E_r, and E_{ad} represent the energy consumed by the microcontroller, the memory, and the A/D converter while executing the program[2]. E_{mc} is a weighted sum of the number of instruction groups—the instructions are grouped according to the number of cycles they take. E_r depends on the number of read/write accesses, the number of refreshes, and the standby power required during the complete execution period. E_{ad} is the sum of the average energy to access the A/D converter and the standby current flowing through the converter during the idle state. The highest energy contributor during their experiments was the energy required for memory accesses.

Jayaseelan et al. [JML06] and Wägemann et al. [WDH+15] proposed worst-case energy models that can provide worst-case energy consumptions of a program at the Instruction Set Architecture (ISA) level. Jayaseelan et al. [JML06] bound the worst-case energy consumption on a simulated processor. However, it is not feasible on the physical hardware as the model is limited by the need to maximize the switching activity on the hardware. Wägemann et al. [WDH+15] did not provide the specifics for the energy model's characterization and concluded that the energy model could lead to significant overestimation.

As the problem of determining worst-case energy consumption is an NP-hard problem, Georgiou et al. [GKCE17] proposed using pseudo-randomly generated data to characterize an ISA energy model, which came close to the actual worst-case. Wegener et al. [WNNYE23] proposed an energy analysis tool called `EnergyAnalyzer` that uses the ARM Cortex-M0 energy model [NGC+22] and static WCET analysis techniques to estimate the worst-case energy consumption of a program. This energy model for ARM Cortex-M0 used in the `EnergyAnalyzer` is based on a linear regression model. The following

[2] A detailed characterization of each term is provided in [KCNL08].

energy model is used within the `EnergyAnalyzer` to calculate the total energy consumption for ARM Cortex-M0 architecture [NGC+22]:

$$\begin{aligned} E = \; & 0.972565030 * \mathbb{C}_1 + 0.652871770 * \mathbb{C}_2 + \\ & 1.031341343 * \mathbb{C}_3 + 1.037625441 * \mathbb{C}_4 + \\ & 1.354953706 * \mathbb{C}_5 + 2.274650563 * \mathbb{C}_6 \end{aligned} \tag{2.4}$$

where $\mathbb{C}_1$ is the counter for the executed instructions without multiplications, $\mathbb{C}_2$ is the counter for RAM data reads, $\mathbb{C}_3$ is the counter for RAM data writes, $\mathbb{C}_4$ is the counter for Flash data reads, $\mathbb{C}_5$ is the counter for the taken branches, $\mathbb{C}_6$ is the counter for the multiplication instructions, and all the constants are their respective energy estimations measured in nJ.

`EnergyAnalyzer` uses this energy model to estimate the energy consumption of each basic block and uses ILP-based techniques to determine the execution path that leads to the worst-case energy consumption. `EnergyAnalyzer` is used in this thesis to perform energy analysis at the compiler level[3].

2.3 Schedulability

The notion of schedulability becomes important when dealing with multi-task systems. A task is a predefined routine executed on the system that fulfills a dedicated purpose. For real-time multi-task systems, each task is assigned a deadline relative to the time at which it becomes ready for execution.

In a multi-task system, task activation plays an important role in determining the system's schedulability. These activation patterns determine how and when the tasks are triggered and executed. There are various task activation patterns, such as periodic, aperiodic, sporadic, event-triggered, etc. In Chapter 7, we consider multi-task systems with periodic tasks during evaluations.

Definition 2.6 (Periodic Tasks)
A periodic task τ_t *is defined as a tuple* (W_t, D_t, AP_t)*, where* W_t *denotes the WCET of* τ_t*,* D_t *is the deadline, and* AP_t *is the activation period of the task* τ_t*.*

The periodic tasks are activated at regular intervals based on a fixed period AP_t. A periodic task τ_t with a period AP_t equal to its deadline D_t is called an implicit deadline task, and a system with all implicit deadline tasks is called an implicit deadline system.

A system is schedulable when all the tasks in a multi-task system are executed without violating its timing constraints. Schedulability analysis is used to verify if all the tasks in the real-time system will meet all timing constraints under any circumstances. Luppold [Lup20] defined system schedulability using the so-called Worst-Case Response Time (WCRT).

[3] A detailed explanation of the `EnergyAnalyzer` is provided in Chapter 3.

Definition 2.7 (Worst-Case Response Time (WCRT))
The Worst-Case Response Time (WCRT) (WR_t) of a task τ_t is the maximum possible interval between any activation of the task and its respective finishing, including all blocking times and interferences of other tasks.

Definition 2.8 (Schedulability)
A system is called schedulable if and only if the WCRT of each task in the system is lower than or equal to the respective task's deadline, i.e.,

$$\forall \tau_t \ : \ WR_t \leqslant D_t \tag{2.5}$$

The task's WCRT plays a critical role in determining whether a system as a whole is schedulable. We can perform scheduling offline or online. Offline scheduling is extremely rigid, fixed at system design time, and does not allow any change during runtime. Online scheduling is flexible, and task execution decisions are made during runtime.

Online scheduling is categorized further into cooperative and scheduler-based scheduling. Cooperative scheduling provides a running task the freedom to decide if and when execution can be handed over to another task, sometimes leading to high-priority tasks missing deadlines and making it incompatible with real-time systems. The scheduler-based approach uses a scheduler within the system that can be hardware-triggered and executed arbitrarily. There exist two main classes of task schedulers: preemptive and non-preemptive.

Definition 2.9 (Preemptive and Non-preemptive Scheduling)
A running task can be interrupted and suspended by another in preemptive scheduling, whereas in non-preemptive scheduling, a currently executing task runs until it is completed [LL73].

Each task, irrespective of whether preemptive or non-preemptive scheduling is assumed, is assigned a priority. The priority of a task is a non-negative integer associated with it. The lower the value of the integer, the higher the task priority. The scheduler decides the next executable task based on the priorities of the entire task set.

During schedulability analysis, task scheduling can be done according to priorities assigned in a fixed or dynamic fashion. Many scheduling algorithms exist at our disposal, which in practice could be used based on different factors like, for example, legacy code, hardware platform restrictions, or safety regulations. In fixed-priority scheduling, as the name suggests, each task is assigned a fixed priority that does not change during runtime. Rate-Monotonic Scheduling (RMS) and Deadline-Monotonic Scheduling (DMS) are examples of fixed-priority scheduling [LL73]. In dynamic priority scheduling, the tasks' priorities are determined dynamically at runtime depending on the runtime state of the system. A task can get preempted by another depending on its

Figure 2.2.: Exemplary von Neumann architecture

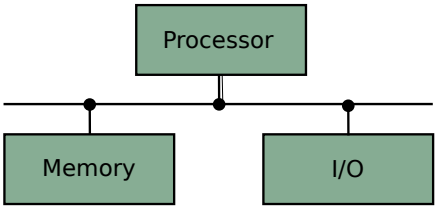
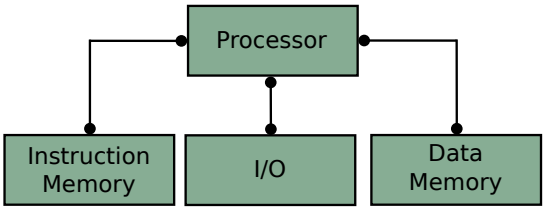


Figure 2.3.: Exemplary Harvard architecture



deadlines. The Earliest Deadline First (EDF) is a popular example of dynamic priority scheduling [LL73]. The priority of each task in a system scheduled under EDF is calculated at runtime according to its absolute deadline. The task with the smallest deadline is assigned the highest priority.

2.4 Embedded Architectures

Microprocessors, memories, input/output devices, and buses largely congregate to describe the underlying embedded hardware.

- A **microprocessor** executes program instructions and performs arithmetic, logical, and control operations.
- A **memory** stores and retrieves data and instructions whenever required by the processor.
- **Input/output devices** serve as an interface between the embedded system and the real world.
- **Buses** connect microprocessors, memories, and input/output devices and establish communication between them.

Embedded architecture is broadly categorized into von Neumann (cf. Figure 2.2) and Harvard (cf. Figure 2.3) architectures.

- A **von Neumann** architecture has a processing unit, memory, and input/output devices connected by a single bus. It uses a single memory space to store instructions and data, for example, ARM7, x86, etc.
- A **Harvard** architecture uses separate memory spaces for instructions and data. It has separate data and instruction buses for simultaneous fetches and transfers, for example, ARM9, MPC860, etc.

2.4.1 Processors

The processor is the "brain" of the embedded system. Embedded processors are generally small, low-power, efficient, reliable, and need to operate for a long time without failure. Processors are largely some combination of an Arithmetic Logic Unit (ALU), registers, a control unit, and internal buses that connect them.

- The **ALU** is a combinational logic circuit that implements math, logical, and comparison operations defined by the Instruction Set Architecture (ISA). ISA refers to the embedded architecture's instruction set.
- **Registers** are a set of flip-flops that temporarily store data or delay signals. A 32-bit processor usually has working registers that are 32-bit wide, i.e., they contain 32 flip-flops. An ISA broadly categorizes registers into general-purpose (storage and manipulation of data) or special-purpose (specified tasks).
- A **control unit** generates timing signals in the processor and controls and coordinates the fetching, decoding, and execution of instructions.

Processors are categorized into different families. Each category has advantages and can be used based on the specific application.

- Application Specific Integrated Circuit (ASIC) processors are smaller and more power- and performance-efficient. However, they are customized for specific applications and cost more.
- Complex Instruction Set Computer (CISC) processors have complex instruction sets that reduce the number of needed instructions and require less memory. However, CISC processors are slow, have more complex designs, and consume more power due to complex instructions.
- Reduced Instruction Set Computer (RISC) processors have simple instructions, faster execution, and consume less power. However, due to the simple instruction set, RISC needs more instructions than CISC. Due to their simple hardware design, RISC processors are more timing-predictable, making them more compatible with real-time applications.

2.4.2 Memory

Memories are storage devices that store instructions and data that the processor uses to process. There are three main categories of memories. A system designer needs to be aware of the differences to use them effectively.

- Read Only Memories (ROM) are non-volatile memories that store data and instructions. ROM is used to store programs that are rarely changed during execution. ROMs are stable and secure, but less reprogrammable. PROM, EPROM, etc., are some of the types of ROM. PROM is a one-time programmable ROM whose content can never be changed once programmed. EPROM is a ROM whose content can be erased and reprogrammed using a special separate device.

- Random Access Memories (RAM) are volatile memories that processors use to store data and instructions. The processor can access any location within RAM directly, and the content stored in the RAM can be changed during execution. Unlike ROM, RAM stores data as long as the power is supplied. Static RAM (SRAM) and dynamic RAM (DRAM) are two main types of RAM. SRAM uses flip-flop circuits to store data. It ensures fast access times and consumes less power. DRAM stores data in capacitors that require periodic refreshing to maintain their charge. DRAM is slower and has a shorter data lifetime compared to SRAM. DRAM is more affordable compared to SRAM.

- Hybrid memories combine features from both RAM and ROM. These devices maintain their data without power, like ROMs, and can be read and written like RAMs during execution. Electrically Erasable Programmable ROM (EEPROM) is like EPROM, except it can be reprogrammed electrically without needing a special device. However, it has a higher cost and longer write cycles than RAM. Non-Volatile RAM (NVRAM) is a modified SRAM that maintains data without power. NVRAM has faster access cycles, but it can be expensive and might have a limited number of write cycles due to wearout.

2.4.2.1 Flash

Flash memory is a high-density, non-volatile, and electrically reprogrammable hybrid memory. The Flash offers fast read access, but writing to it can take significantly longer. Due to its high density, low cost, and fast read performance, Flash is largely used in embedded systems to store applications that can be moved to other volatile memories during application startup.

2.4.2.2 Cache

A cache is a small and fast memory between the processor and the Flash. An embedded architecture can host different levels of caches; L1 and L2 are the most common SRAM caches; L1 are faster than L2; L2 are larger than L1.

If a processor tries to read data from memory, the L1 cache is checked first. If the data exists in the L1 cache, we call it a cache hit; if not, it is a cache miss. In case of a cache miss, L2 cache, Flash memory, etc., is checked hierarchically for the desired data. Once the data is found, it is typically copied into the

cache, anticipating future needs. Data from the cache is removed or evicted when space needs freeing to copy new data to the cache.

Caches can autonomously decide when and how to load data into themselves without requiring explicit interference from the CPU. Caches implement replacement policies like the Least Recently Used (LRU), First-In-First-Out (FIFO), or Random Replacement policy to decide which data can be evicted during runtime. These policies help optimize cache performance without direct control from the software or CPU.

As the content in the cache depends on the runtime behavior of the system and the replacement policy used by the caches, caches are less predictable and difficult to analyze for the system's real-time behavior. In the past, cache locking techniques have been exploited to control cache behavior during runtime [PD02, PKM+12]. The cache locking technique, as the name implies, places code or data within the caches at compile time and locks it. As the content within the cache does not change here during execution, its behavior is predictable. However, as we lose the benefits of using caches in such cases, we could use SPMs instead.

2.4.2.3 ScratchPad Memory (SPM)

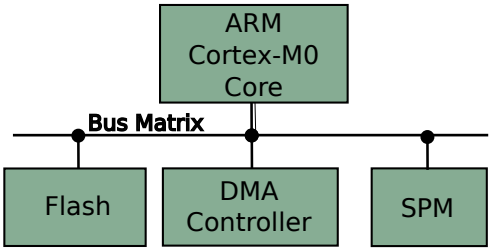
ScratchPad memory (SPM) is a small high-speed SRAM directly connected to the processor. Technically, SPMs and caches are implemented with the same technology, and the system designer can mostly configure the processor to use this memory as an SPM or a cache. SPM overcomes the unpredictable nature of caches by keeping its content fixed. Therefore, SPMs are largely used in the real-time domain to increase the performance or energy efficiency of the code.

As SPMs cannot evict their content autonomously, a software-controlled approach could be used to predictably rewrite the content within SPM during execution. In this thesis, we will use dynamic SPM allocation to replace the content in SPM during runtime in a controlled and predictable manner, thus utilizing SPMs to their full extent.

2.4.3 Direct Memory Access (DMA)

A Direct Memory Access (DMA) controller transfers data between embedded components without involving the processor. The DMA controller can transfer data between the main memory and peripherals and between memories during runtime. An Advanced High-performance Bus (AHB) matrix connects the DMA controller to the processor and the memories. Figure 2.4 represents an ARM Cortex-M0 processor with a DMA controller, Flash, and SPM connected via a bus matrix. Generally, commercial DMA controllers are equipped with multiple DMA channels that can be programmed for simultaneous data transfers. However, in Chapter 9, we focus solely on memory-to-memory DMA transfers from Flash to SPM. Memory-to-memory DMA transfer requires three steps: arbitration for bus access, address computation, and data transfer.

Figure 2.4.: ARM Cortex-M0 architecture



- **Arbitration for the Bus Access:** It is ensured that bus access is granted in a controlled and fair manner, as multiple devices, like CPU, DMA, etc., may need to use the system bus simultaneously.

- **Address Computation:** The DMA controller needs to set up the source and destination addresses in its registers, and depending on the type of transfer, the DMA controller may need to increment or decrement the addresses after each byte is transferred. In a memory-to-memory type of transfer (our case), the source and destination addresses are typically incremented after each transfer to handle sequential data blocks.

- **Data Transfer:** The data is actually moved from the source address to the destination address.

The DMA controller can transfer data in two different modes: burst and arbitration mode.

- **Burst Mode:** The DMA controller transfers the entire object without CPU interruptions, i.e., the CPU is stalled for the whole duration of the burst if it wants to access the memory.

- **Arbitration Mode:** The DMA controller transfers the object over the bus matrix whenever the CPU is executing instructions and not accessing memories. If the CPU needs access to the bus matrix to fetch instructions or load and store data in memory, bus arbitration takes place, granting the CPU access to the bus. The bus matrix uses scheduling to divide bus bandwidth between the processor and the DMA controller. In arbitration mode, we assume that the bus matrix employs round-robin arbitration [SPG91] to ensure that at least half of the system bus bandwidth is available for the CPU. Bus arbitration only occurs when two masters need to access the same slave memory.

In this thesis, we consider the processor and the DMA controller as the masters and Flash and SPM as the slave memories. The DMA controller can independently transfer data from Flash to SPM, but the processor needs to

initiate the DMA transfer. Therefore, bus arbitration occurs when the CPU and the DMA controller want to access the same memory.

To ensure that the first DMA transfer is over before initiating the second, we can use polling or interrupt mechanisms to ensure the channel is free.

- **Interrupt Mechanism:** The DMA controller can generate an interrupt when a transfer is complete.
- **Polling Mechanism:** The CPU repeatedly checks a status flag in the DMA controller's status register.

Before initializing the next DMA transfer, we use the polling mechanism in this thesis (in Chapter 9) to check if the previous DMA transfer is complete. Moreover, we use the polling mechanism to ensure the DMA transfer is complete before executing an object from SPM.

2.4.4 ARM Cortex-M0

The ARM Cortex-M0 [ARM09a, ARM09b] is a 32-bit RISC, small, and power-efficient embedded processor with a 3-stage pipeline. The ARM Cortex-M0 processor implements the ARMv6-M Thumb instruction set (16-bit) and a number of 32-bit instructions from Thumb 2. The ARM Cortex-M0 instructions comprise the move, load, store, logical, add, multiply, etc., operations. It is a von Neumann-based processor. Therefore, data load or store operations and instruction fetching cannot simultaneously occur.

In this thesis, we use the STM32F0xx microcontroller [STM22a] implementation of ARM Cortex-M0 as a reference (cf. Figure 2.4). It constitutes a Cortex-M0 core, SRAM (SPM), Flash, AHB-lite buses, and a DMA controller. The instruction memory, data memory, input/output ports, and registers are within a linear 4GB address range. A bus matrix manages access arbitration using round-robin arbitration between the Cortex-M0 core, the DMA, and memories. According to the STM32F0xx microcontroller manual [STM22b], the number of cycles required to transfer data using the DMA controller is:

$$t_s = t_A + t_{RD} + t_{WR} \tag{2.6}$$

where t_A is the arbitration time, t_{RD} is the read access time, and t_{WR} is the write access time. In this thesis, αT_F and αT_S represent the access latencies for the Flash memory and the SPM. For memory-to-memory (Flash-to-SPM) transfer, t_{RD} is the Flash's read access latency, and t_{WR} is the SPM's write access latency, i.e., in this case, $t_{RD} = \alpha T_F$ and $t_{WR} = \alpha T_S$.

A system designer can invoke the DMA controller to relieve the CPU of data transfer duties. In this thesis, we use ARM Cortex-M0 as the underlying architecture while performing analyses and optimizations during evaluations. During the evaluations conducted in this thesis, we assume $\alpha T_F = 6$ cycles and $\alpha T_S = 1$ cycle for the underlying ARM Cortex-M0 architecture.

3

WCET-aware C Compiler (WCC)

Hardware and software are two sides of the same coin that play critical roles in developing embedded systems. Having the best hardware specifications and poor-quality software defeats the purpose of investing in the said hardware. Compilers can deliver high-quality embedded software that meets stringent constraints posed by the hardware resources and the underlying embedded application. Standard compilers include a variety of optimizations, primarily aiming at optimizing software's average-case behavior. However, real-time embedded systems encounter WCET and energy constraints. As the average-case, worst-case, and energy-intensive behavior of software can depend on different regions of the software, optimizing for the average case is not sufficient for real-time systems.

Current embedded software development is riddled with human intervention. A system designer often models the system based on its requirements using model-based graphical tools, generates code, and optimizes and compiles it. As standard compilers do not have the notion of worst-case- and energy-behavior of underlying hardware integrated within them, the optimizations are not fruitful against these objectives. Furthermore, if the compiled code violates the set requirements, the system designer has to manually tweak the system model or the generated code and repeat the process. It is a highly inefficient process with the possibility of introducing human errors. If a compiler can understand the system requirements and analyze and optimize code for real-time and energy constraints, we can automatically optimize code efficiently and remove the need for human intervention.

The WCET-aware C Compiler (WCC) [FL10] is a C compiler framework that performs analyses and optimizations targeted specifically towards real-time embedded systems. WCC is integrated with a static WCET analysis tool called `aiT` [Abs24] and an energy analysis tool called `EnergyAnalyzer` [WNNYE23]. They analyze C code automatically and provide the necessary data to perform WCET- and energy-aware optimizations without human intervention. Without activating WCET and energy analyses, WCC can also compile code as standard C compilers like `GCC` [Sta08] and `LLVM` [LA04].

WCC supports various single-objective optimizations that target the program's WCET or energy consumption. It also supports a couple of multi-objective optimizations that target the high-level Intermediate Representation

(IR) to optimize WCET and energy consumption simultaneously. This thesis extends the WCC infrastructure towards multi-objective optimizations that target the assembly-like low-level IR. The proposed low-level optimizations exploit the architecture's memory hierarchy to find trade-offs between the objectives at compile time.

WCC supports ARM7TDMI from ARM and TC1796 and TC1797 processors from Infineon TriCore. It also supports an artificial ARM7TDMI multi-core system for scientific purposes [Kel15]. WCC is extended to support ARM Cortex-M0 and Leon3 architectures [Tea18] during the `TeamPlay` project [RBE+23]. We use the ARM Cortex-M0 fork of WCC to compile code and demonstrate the approaches proposed in this thesis.

This chapter is organized as follows: Section 3.1 presents the internal structure of WCC. Section 3.2 describes WCET and energy analyses, respective analyzers, and their integration with WCC. Section 3.3 briefly describes the WCC-internal schedulability analyzer used for schedulability analysis in Chapter 7.

3.1 WCC Infrastructure

WCC's overall internal structure is depicted in Figure 3.1. WCC consists of components that resemble a typical optimizing compiler and some components that make WCC a specialized compiler that enables WCET- and energy-aware optimizations.

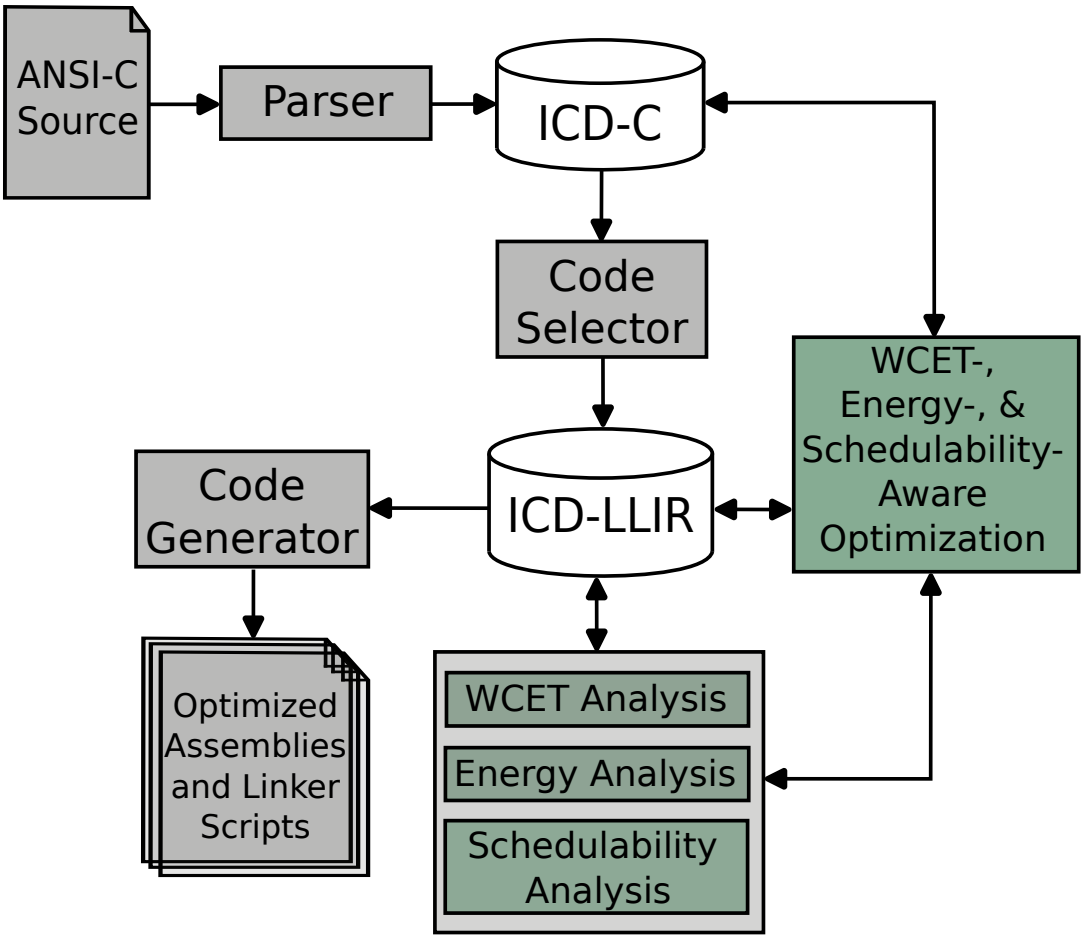
- **Parser:** The parser is fully compliant with ANSI C and performs lexical, syntax, and semantic analyses. It accepts several C source files within a single compiler run and creates a high-level IR called ICD-C.

- **ICD-C:** The ICD-C [Dor09] framework is a data structure that provides an architecture-independent IR from C. At the ICD-C level, WCC can perform architecture-independent high-level optimizations like function inlining, function specialization, etc.

- **Code selector:** The code selector translates ICD-C into a low-level IR called ICD-LLIR.

- **ICD-LLIR:** ICD-LLIR [Dor05] is an architecture-specific assembly-like code representation. It is used as a data structure providing a retargetable low-level IR for the compiler backend. At the ICD-LLIR level, WCC can perform architecture-dependent low-level optimizations like register allocation, SPM allocation, etc. In this thesis, we propose and implement low-level multi-objective static and dynamic SPM allocation at the ICD-LLIR level.

Figure 3.1.: WCC infrastructure: The optimization and analysis parts, represented in green, differentiate the WCC compiler from general modern compilers.



- **Code generation:** The code generation finally emits valid assembly code and tailored linker scripts from this architecture-specific LLIR.

- **Analyzers:** WCC has integrated WCET, energy, and schedulability analyzers, enabling it to perform respective analyses. WCET and energy analyzers are integrated within WCC at the ICD-LLIR level as they require low-level architecture information and executables. Similarly, the schedulability analyzer is also at the ICD-LLIR level, as it requires WCET analysis results as input.

- **Optimizations:** WCC uses respective analyzers to evaluate the code for respective objectives while performing optimizations. WCC supports ILP-based single-objective and metaheuristic algorithm-based multi-objective optimizations both at ICD-C and ICD-LLIR levels.

This section is further organized as follows: Subsection 3.1.1 discusses memory layout-related infrastructure within WCC. Subsection 3.1.2 presents *flow fact* annotations used within WCC that help while performing WCET and energy analyses. Subsection 3.1.3 describes the jump correction needed to rectify the control flow graph if code is moved from one memory to another during memory allocation.

3.1.1 Memory Hierarchy

Embedded memories have a big impact on the non-functional properties, like execution time and energy consumption of a program. Executing code and accessing data from fast or slow memory can make a huge difference in the real-time performance of the software. WCC generates code for different embedded architectures that host their specific ISAs and memory hierarchies. However, the memory subsystems can differ for different microcontrollers with the same embedded cores. For example, STM32F03x and STM32F05x are microcontrollers that host an ARM Cortex-M0 core, but STM32F03x has 4KB SRAM and 32KB Flash, whereas STM32F05x has 8KB SRAM and 64KB Flash [STM22a]. Therefore, the WCC framework provides a system designer with a means of specifying a detailed memory hierarchy. Furthermore, WCC exploits this memory hierarchy during memory-aware optimizations to minimize the program's non-functional properties.

Listing 3.1: Exemplary Memory Specification within WCC

```
# Flash (FLASH; Code)
[FLASH-CODE]
origin          = 0x08000000
length          = 0x8000       # 32K
attributes      = RWXA         # read/write/execute/allocate
sections        = .text
cycles          = 6
memory_type     = FLASH
```

WCC provides a simple text interface through a memory layout file to specify an architecture-dependent memory hierarchy. The memory hierarchy file provides specifications for the different regions of a processor's physical memory. The WCET and energy analyzers in WCC use the memory hierarchy defined within this memory layout file during analysis. For each physical memory region, a system designer can define the following attributes:

- `origin` and `length`: The memory region's start address and its absolute length.
- `attributes`: The type of allowed memory accesses, like read, write, execute, etc.

- `sections`: The assembly-level sections serve as containers for program code, data, constants, etc. The sections are allowed to be mapped to different memory regions.
- `cycles`: The cycles elapsed until the content of the memory is fetched or written.
- `memory_type`: The type of the physical memory.

Listing 3.1 shows an exemplary excerpt of the memory layout specification for the Flash memory region for ARM Cortex-M0 within WCC. For caches, a system designer can also specify attributes like absolute sizes, line sizes, or associativity. We use the *sections* attribute extensively during static and dynamic memory allocation-based optimization within WCC—`section` defines the physical memory region to which the assembly code or data will be allocated.

3.1.2 Flow Fact Specifications

A program's non-functional properties are determined by its control flow. Before defining a Control Flow Graph (CFG), we first define a basic block:

Definition 3.1 (Basic Block)
A basic block $b = (i_1, \ldots, i_l)$ *is an instruction sequence of maximal length such that the basic block* b *is entered only via its first instruction and is left only via its last instruction [Muc98].*

Definition 3.2 (Control Flow Graph)
The Control Flow Graph (CFG) of a program is a directed graph that represents all possible execution paths of the program, where a node of a CFG represents a basic block and an edge represents the transfer of the execution from one basic block to another.

The CFG always has a start node with no predecessors. This start node can either be the entry basic block of a function or the entry basic block of the entire program, from where execution is assumed to begin. Not all basic blocks in a CFG are guaranteed to be executed during a program's run—for example, basic blocks within conditional branches like if-else may not be traversed. A node in the CFG (i.e., basic block b_i) is said to dominate another node b_j if every path from the start node b_1 (the entry basic block) to b_j passes through b_i. Every basic block in the CFG, except the start node b_1, is dominated by at least one basic block other than itself. If basic block b_i dominates b_j, then b_i is called a dominator of b_j [App04].

As mentioned in Section 2.1, software determinism is important while performing static analysis. Arbitrary loop counts and unrestricted control flows can hamper WCET and energy calculations. Since real-time systems need precise WCET and energy analyses, a system designer needs to annotate

the program's loops and any recursive cycles in the CFG. These user-provided annotations are generally called *flow facts*.

Definition 3.3 (Flow Facts)
Flow facts describe the set of possible execution paths of a program. Flow facts that exist due to the structure and semantics of a program code are called implicit flow facts. Flow facts provided by a user are called annotated flow facts [Kir03].

The static WCET and energy analyzers within WCC require the following kinds of *flow facts* to perform safe and precise analyses.

- Loop iteration counts
- Recursion depths
- Execution frequency of an instruction relative to the others

WCC is a C compiler and fully supports source code-level *flow facts* employing ANSI C pragmas. Loop bound annotation is one of the most important *flow fact* annotations required for precise WCET and energy analyses. Loop bounds specify upper and lower limits on the iteration counts of regular loops like for, do-while, and while loops. This annotation is for single-entry loops with well-defined termination conditions. Listing 3.2 shows an exemplary loop bound annotation employed using an ANSI C pragma for a for-loop executed 100 times.

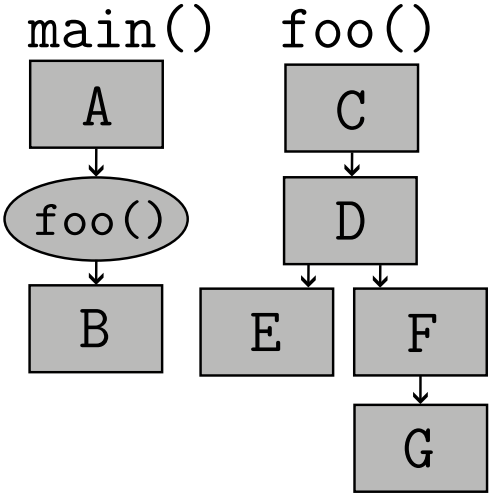
Listing 3.2: Exemplary loop bound annotation

```
1 int res = 0;
2
3 _Pragma( "loopbound min 100 max 100" )
4 for ( int i = 1; i <= 100; i++ ) {
5   res += i;
6 }
```

Simple Loop bound annotations are inapplicable for irregular loops. WCC uses *flow restriction* annotations to annotate dependent loops, multi-entry loops, loops without explicit termination conditions, or loops with *goto* statements. Flow restriction annotations relate the execution frequency of one C statement to another. A system designer can use an auxiliary annotation called a *marker* that attaches an identifying string to mark a C statement.

A complex flow restriction between different C statements is defined using the *marker* identifiers. Flow restriction annotations allow a system designer to define linear dependencies between the markers. Listing 3.3 shows an exemplary flow restriction annotation example. It states that the execution frequency of *Statement B*, i.e., marker *inside*, is at most 55 times larger than *Statement A*, i.e., marker *outside*.

Figure 3.2.: Exemplary CFG with two functions `main()` and `foo()`



Listing 3.3: Exemplary flow restriction annotation

```
1  _Pragma( "marker outside" )
2  Statement A;
3
4  for ( int i = 1; i <= 10; i++ ) {
5    for ( int j = i; j <= 10; j++ ) {
6      _Pragma( "marker inside" )
7      Statement B;
8    }
9  }
10 _Pragma( "flowrestriction 1*inside <= 55*outside" )
```

WCC automatically identifies the *flow facts* internally at the ICD-C level, translates them, and transfers them to the ICD-LLIR level. Within WCC at the LLIR level, the *flow facts* information is passed to `aiT` and `EnergyAnalyzer`, and they use these annotations for precise WCET and energy analyses.

3.1.3 Jump Correction

Static and dynamic memory allocation techniques move code from one memory to another either during compile-time or runtime. Allocating code from one memory to another can break the program's CFG, make previously reachable code unreachable, or invalidate previously valid jumps and branches. Therefore, we use jump correction within WCC to fix the CFG, reachability, and broken jumps and branches issues while performing memory allocation-based optimizations.

Figure 3.2 shows an exemplary CFG with two functions (`main()` and `foo()`), `main()` has two basic blocks—`A` and `B`—, basic block `A` calls function `foo()`, and function `foo()` has five basic blocks `C` to `G`. Next, Figure 3.3 shows an exemplary basic block allocation to either the Flash or SPM for the CFG from

Figure 3.3.: Basic block assignment for the exemplary CFG from Figure 3.2, where C and F are placed in SPM

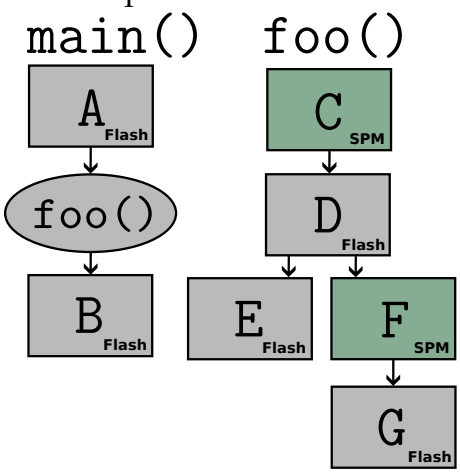


Figure 3.2. In the example allocation, basic blocks C and F from function foo() are assigned to SPM, and the rest remain in the Flash.

Jump correction is needed in cases where the last instruction of a basic block is no longer able to reach the next basic block due to memory allocation. Generally, jump, call, and branch instructions are called control transfer instructions, as they alter the normal sequential flow of program execution. In the case of PC-relative control transfer instructions, the distance covered by these instructions decides their validity. In the processor's memory layout, the distance between the Flash and SPM memory regions may exceed the maximum distance that can be covered by such control transfer instructions. In the case of fall-through basic blocks—where their last instruction is not a control transfer instruction—, the CFG is broken due to allocation, and we need to add new jump instructions to fix the CFG.

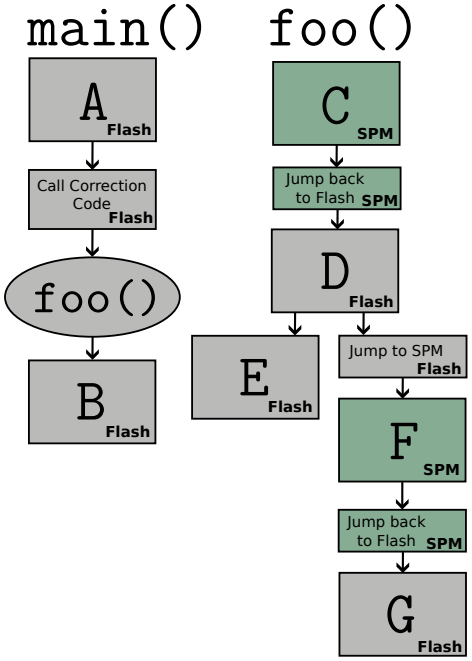
In Figure 3.3, the basic block C is the first basic block of the function foo(). So, the call for foo() in the function main() in the Flash might not reach basic block C. Similarly, allocating basic block F in SPM can break the CFG, and jump corrections between D and F and F and G might be needed.

Figure 3.4 shows places within the CFG where code transformations might be needed after allocating C and F. We need to decide the type of code transformation required to accommodate the allocation of basic blocks to SPM, depending on the last instruction of the basic block. Oehlert et al. [OLF16] discussed practical challenges incurred due to SPM allocation and presented modeling aspects of compiler-level jump correction required for SPM allocation.

Listing 3.4: Exemplary original function call code

```
1 .section .text  /*Code in .text section is placed in the Flash*/
2 main:           /*First basic block A for main() function*/
3   ...
4   bl.w foo;     /*Call instruction for function foo()*/
```

Figure 3.4.: Exemplary code transformations for SPM allocation



Listing 3.4 gives an example assembly-level code for the first basic block `A` of the function `main()` according to Figure 3.2. Basic block `A` calls function `foo()` using a label-based branch and link (`bl.w`) instruction. The assembly code indicated in the listing is for the ARM Cortex-M0 architecture.

If the basic block `C`—the first basic block of function `foo()`—is allocated to SPM, the `bl.w` instruction cannot reach `C` in SPM from Flash. So, we perform jump correction and fix the function call as shown in Listing 3.5. The label-based branch and link instruction is replaced by a load instruction (`ldr`) and a register-based branch and link call (`blx.n`). We also need to add a literal pool that can hold the target addresses.

Listing 3.5: Exemplary function call code after jump correction

```
1  .section .text
2  main:
3    ...
4    ldr.n r1, .temp_foo_callBB; /*Load foo()'s target SPM address
5                                 *to register r1*/
6    /*Call target: foo*/
7    blx.n r1;
8
9  .temp_foo_callBB:            /*New literal pool created*/
10   .word foo;
```

ARM compilers can put data objects in the code sections normally reserved for instructions. Literal pools are such basic blocks that hold data constants that cannot be directly encoded in instructions due to the limited range of immediate

values that instructions can support. For example, in Listing 3.5, the `ldr.n` instruction cannot hold the target address of `foo()` directly as an immediate value. So, the literal pool `.temp_foo_callBB` holds the function `foo()`'s target address, and the `ldr.n` instruction loads the word from `.temp_foo_callBB` and writes it to the register `r1`.

Listing 3.6 gives an example assembly code for the first basic block `C` of the `foo()` function, according to Figure 3.2. Basic block `C` is a fall-through basic block, i.e., it does not have a branch, return, call, etc. instruction at its end. The example assembly code indicated in the listing shows a move instruction at the end of `C`, and after its execution, basic block `D` will be executed.

Listing 3.6: Exemplary original fall-through code

```
1  .section .text
2  foo:            /*First basic block C for foo() function*/
3    ...
4    mov.n r0, 1;
5
6  .D:             /*Both basic blocks C and D are in Flash*/
7    ...
```

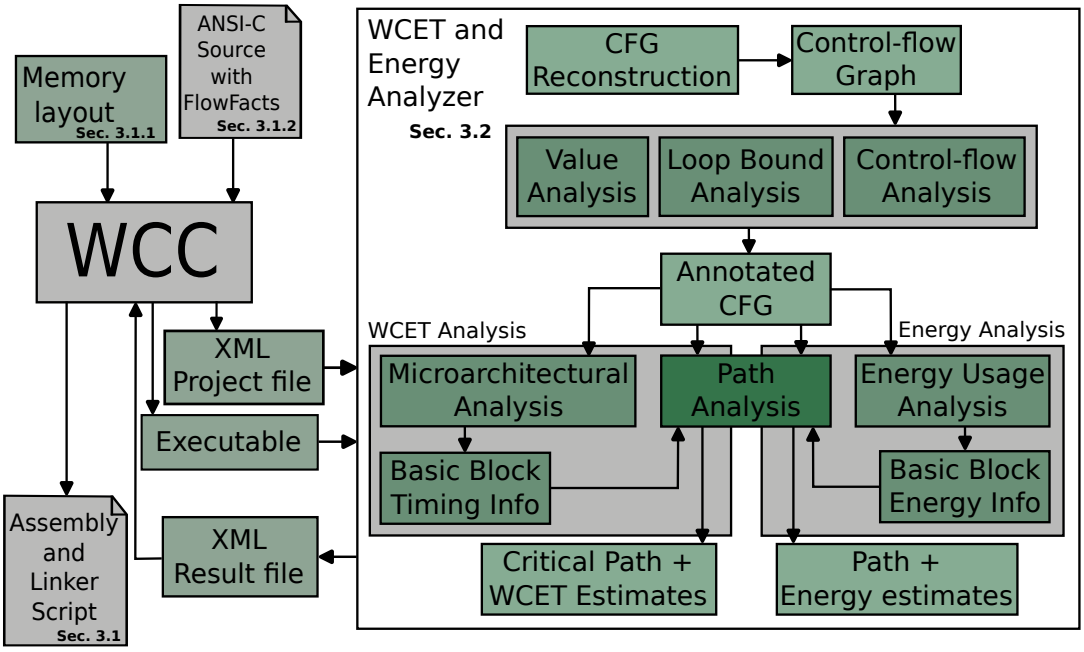
If `C` is placed in SPM according to the allocation shown in Figure 3.3 and basic block `D` remains in the Flash, the CFG breaks, and there is no instruction for the execution to return to `D` in Flash from `C` in SPM. So, we perform jump correction and fix the fall-through code, as shown in Listing 3.7. We add a load instruction that loads the target address of `D` from Flash into the register `r1` and a branch instruction (`bx.n`) that is able to branch to an address stored in the register. We also add a literal pool (`.temp_D_jumpBB`) that holds the target address of the basic block `D`.

Listing 3.7: Exemplary fall-through code after jump correction

```
1  .section .text_spm /*Code in .text_spm section is placed in SPM*/
2  foo:               /*First basic block C of foo() is in SPM*/
3    ...
4    mov.n r0, 1;
5    ldr.n r1, .temp_D_jumpBB; /*Load .D's target Flash address to
6                               *register r1*/
7    /*Branch target: .D*/
8    bx.n r1;
9
10 .temp_D_jumpBB:             /*New literal pool created*/
11   .word .D;
12
13 .section .text
14 .D:                         /*Basic block D in the Flash*/
15   ...
```

In these two example jump corrections, we assume that the register `r1` is free. However, if no free register is found, we might need to perform register spilling, which can further increase the code size and incur extra WCET and energy

Figure 3.5.: WCET and energy analyses framework



costs. Similar jump corrections are also performed for cases with conditional branches, unconditional branches, multiple successors, etc. However, for the sake of brevity, we skip the discussion of many such cases that are considered during jump correction.

3.2 WCET and Energy Consumption Analyzers

In this thesis, we use AbsInt's static WCET analyzer called `aiT` [Abs24] and a static energy consumption analyzer called `EnergyAnalyzer` [Tea20]. `aiT` supports WCET analysis for TriCore, ARM, and Leon architectures within WCC. `EnergyAnalyzer` supports energy analysis for ARM (specifically Cortex-M0) and Leon architectures within WCC. The WCET and energy analysis results are attached at the basic block, function, LLIR, and task level within WCC.

Figure 3.5 shows AbsInt's WCET and energy analyzer internal framework and its integration with WCC. A system designer can provide WCC with an ANSI C program file that is annotated with *Flow Facts* (as described in Subsection 3.1.2). The system designer can also provide a memory layout file that defines various attributes of the underlying architecture's memory hierarchy (as described in Subsection 3.1.1).

WCC generates an XML project file that contains links to the annotation file and the executable being analyzed, the type of analysis, and the target

architecture. The annotation file contains details about the target architecture configuration, AIS annotations [Abs], and memory access patterns. The analyzer takes the XML project file and the executable as input with all AIS and architecture-specific details. After the analysis, the analyzer generates an XML output result file with all the WCET and energy analysis results. WCC retrieves WCET and energy analysis results from the XML result file at the LLIR level and annotates these results at the basic block, function, LLIR, and task level. These results are used within WCC for different WCET- and energy-aware optimizations.

The analyzer uses the input executable to reconstruct the program's CFG. Each node of the CFG represents the basic block from the WCC compiler's LLIR, and for each node, WCET and energy usage are estimated within the analyzer. After CFG reconstruction, value analysis safely approximates the values of registers and memory cells. It checks assertions about the reachability of the code to prove the absence of illegal memory accesses.

Loop bound analysis uses value analysis to determine upper and lower bounds for the number of iterations of loops. Control flow analysis also uses value analysis results to determine execution frequencies of the paths, eliminate infeasible paths, and identify contradictory conditions. All the above-mentioned base analyses are needed to perform both WCET and energy analyses.

The WCET analysis within AbsInt's analyzer requires a microarchitectural analysis. It uses a mathematical model of the target processor to derive timing bounds for each instruction, integrating the intrinsic behavior of the pipeline and the memory hierarchy. The WCET analyzer uses results from microarchitectural analysis, CFG, and the basic block timing information to carry out a path analysis.

The path analysis is a generic analysis and does not depend on the underlying architecture or the type of analysis, i.e., WCET and energy analysis. The path analysis collects the WCET estimates at the control-flow edges for all basic blocks and computes a WCET estimation for the entire program using an ILP model. The ILP model is solved to determine a path within the CFG with the longest execution time. This path is the Worst-Case Execution Path (WCEP) of the program, and the execution times of the basic blocks along the WCEP determine the final WCET of the program.

Definition 3.4 (WCEP)
The WCEP of a program is defined as the execution path through a program's CFG, which has the maximal WCET [FL10].

The WCET analyzer also computes how often a piece of code is executed in the worst case. This execution frequency computed by the WCET analyzer in the worst-case is called the Worst-Case Execution Count (WCEC) for each basic block, and they are annotated to the basic blocks within WCC [FK11].

Definition 3.5 (WCEC)
The WCEC of a basic block is the number of times a basic block is executed in the worst-case scenario.

The energy analysis within AbsInt's analyzer (`EnergyAnalyzer`) works on similar lines as the WCET analysis explained above, with a few changes. The energy analyzer employs a microarchitectural energy model to estimate the energy consumption of each instruction and subsequently for each basic block. As mentioned in Section 2.2, the energy model for ARM Cortex-M0 used in the `EnergyAnalyzer` is based on the linear regression model from Equation 2.4. The CFG information, together with basic block energy information, is used to search the program's path with the highest energy consumption. Similar to the WCET analysis, the energy analysis uses path analysis to estimate the highest energy consumption of the program. This path is the Worst-Case Energy Consumption Path (WCECP) of the program.

Definition 3.6 (WCECP)
The WCECP of a program is defined as the execution path through a program's CFG, which has the maximal energy consumption (this definition is based on the concept of the program path for worst-case energy consumption from [WDH+15]).

Moreover, similar to `aiT`, the `EnergyAnalyzer` also provides the Worst-Case Energy Consumption Count (WCECC) for each basic block, i.e., the execution frequency for each basic block in the case that leads to the worst-case energy consumption.

Definition 3.7 (WCECC)
The WCECC of a basic block is defined as the number of times a basic block is executed in the scenario that leads to the worst-case energy consumption.

In this thesis, we utilize `aiT` and `EnergyAnalyzer` to perform WCET and energy analyses and calculate respective objective values for multi-objective optimization. During evaluations, all the WCET and energy analyses are conducted using v21.04i versions of `aiT` and `EnergyAnalyzer`.

3.3 Schedulability Analyzer

In this thesis, we use an internal schedulability analyzer [Lup20] integrated within WCC to perform schedulability analysis. A precursor for performing schedulability analysis is WCET analysis. WCET analysis is performed for each *entry point* of a multi-task system, and results from these WCET analyses are used to determine the schedulability of the system.

WCC provides a system designer with different scheduling algorithm options at their fingertips. The system designer can choose, using command

line options, between EDF, DMS, RMS, etc., and the schedulability analyzer uses the selected algorithm to analyze the multi-task system. In this thesis, we schedule a multi-task system with strictly periodic tasks using the EDF algorithm with deadlines as their periods. Such a system is schedulable under EDF if the following condition holds [LL73]:

$$l_s = \sum_{\tau_t \in \Gamma} \frac{W_t}{AT_t} \leqslant 1 \tag{3.1}$$

where l_s is the system load, Γ is a set of all tasks of the multi-task system, $\tau_t \in \Gamma$ is a periodic task, t is a non-negative task's unique identifier number, and W_t is the WCET of the task. AT_t is the fixed activation period of the task τ_t, and the task τ_t has an implicit deadline $D_t \equiv AT_t$. Equation 3.1 implies that the system is schedulable if and only if its load l_s is lower than or equal to 1.

If a multi-task system is not schedulable, the schedulability analyzer can provide conditions under which the system becomes schedulable. WCC provides the system designer with a set of tasks that, when removed from the multi-task system, make the system schedulable. The schedulability analyzer uses an ILP-based model to perform the analysis, and the t^{th} task's runtime behavior is denoted by an ILP integer variable q_t. q_t is modeled as follows:

$$q_t = \nu_t W_t \tag{3.2}$$

The ILP solver can freely set the value of the binary variable ν_t to 1 or 0. If the multi-task system is unschedulable, the ILP solver can freely set $\nu_t \equiv 0$ and choose to remove the t^{th} task from the task set virtually. The ILP's objective function for a task set Γ with T tasks is:

$$\max \sum_{t=1}^{T} \nu_t \tag{3.3}$$

Due to this objective function, the ILP solver strives to minimize the number of tasks removed from the task set, and the schedulability analyzer ensures that the resulting system is schedulable. In this thesis, we use this feature to define the schedulability objective in Chapter 7 as a non-binary number.

4

Multi-Objective Fundamentals

Optimization is crucial for any development process; an embedded system is no exception. Optimization problems are divided into different categories based on the nature of the objective function and the constraints. An objective function could be continuous, discrete, linear, non-linear, convex, non-convex, etc. It can have one exclusive, multiple local, global, or indifferent optima. The optimization problem can have linear or non-linear constraints or can be unconstrained. The approach to solving the underlying optimization problem changes depending on the objective function and the constraints.

A general equation describing either WCET or the energy consumption of any program for any underlying architecture is unknown. The objective values for WCET and energy consumption can be estimated using respective analyses on individual programs for the underlying architecture. Therefore, to optimize such objectives, we need to find optimal solutions from a finite number of possible solutions, making it a discrete optimization problem.

Embedded systems incur various constraints due to the functional and temporal aspects of the system. A system designer might have to juggle multiple objectives and constraints while developing embedded systems. A fundamental divide between optimization problems and the approaches to solve them occurs based on the number of objectives under consideration.

In this chapter, we discuss the fundamentals of optimizations—mainly focusing on multi-objective optimization. Section 4.1 briefly discusses single-objective optimizations. Section 4.2 presents some concepts related to ILPs. Section 4.3 describes the principles of multi-objective optimization used in the upcoming chapters. Section 4.4 describes the metaheuristic algorithms used to solve multi-objective optimization.

4.1 Single-Objective Optimization

A single-objective optimization, as the name suggests, focuses on minimizing or maximizing one objective for a system. Based on the inherent nature, the size, and the constraints applied to the problem, it can be complicated and time-consuming to solve single-objective optimizations. However, we can easily establish a quality relation between two solutions in a single-objective optimization—for the minimization problem, the smaller the value of the

objective function, the better, and vice versa for the maximization problem. ILP-, heuristics-, and metaheuristics-based approaches can be used to solve such problems.

In general, ILPs are deterministic but NP-hard and can lead to an exponential increase in computational complexity. Contrary to ILP's rigor, heuristics- and metaheuristics-based approaches can be non-deterministic for complex optimization problems. Heuristics can be deterministic for a fixed rule-based approach—if no randomness is involved. However, metaheuristic algorithms are designed to rely on randomized or probabilistic decision-making while exploring the search space.

Furthermore, contrary to ILPs, heuristics- and metaheuristics-based approaches do not guarantee the optimality of obtained solutions. They are generally used to solve NP-hard problems, where an exhaustive search for the optimal solution is computationally infeasible. However, they are practical alternatives for solving large problems and provide solutions for real-world problems in polynomial time.

Single-objective optimizations have been extensively researched and applied in the field of embedded hardware and software development. Either ACET or energy consumption minimization problems have been a hot topic in embedded development for a long time. Researchers started focusing on WCET-aware optimizations for real-time systems in the 90s. Many WCET-, ACET-, or energy consumption-based single-objective optimizations have been exploited in the context of compilers for real-time embedded systems. WCC supports ILP-based single-objective optimizations that aim to minimize the WCET of a program [FK09, FSS11].

In upcoming chapters, we compare the proposed multi-objective optimizations with single-objective ILP-based optimization previously implemented within WCC. In Chapters 8 and 9, we implemented some state-of-the-art single-objective optimizations within WCC that try to minimize WCET or energy while performing compiler-level dynamic allocation and DMA-aware function-level dynamic allocation. In Chapter 9, we propose a DMA Call Placement Optimization (DCPO), which is a single-objective optimization that aims to minimize the total execution time required for DMA transfer from one memory to another in arbitration mode. We solve the DCPO problem using ILPs and two metaheuristic algorithms and compare their solution quality to determine the better-performing approach for this problem.

4.2 Integer Linear Programming (ILP)

Integer Linear Programming (ILP) is a powerful optimization model that can optimize a linear function subject to linear constraints. Let $y \in \mathbb{Z}^d$ be a decision variable vector for an ILP model, where d is the total number of

decision variables and y_i, $\forall i = \overline{1, d}$[1] can only take an integer value. Then, an ILP-based optimization problem can be formulated as follows:

$$\min_{y} a^T y \quad \text{subject to} \quad \mathbb{B}\, y \leqslant c \tag{4.1}$$

where $a^T y$ is the objective function that needs to be minimized, and $\mathbb{B}\, y \leqslant c$ is the set of constraints applied on the decision variable vector y. a is a vector of coefficients multiplied by the decision variable vector, and $\mathbb{B}$ is an $m \times d$ matrix of coefficients. m is the total number of constraints, and c is a vector of bounds on the constraints. While the decision variables must be integer values, the coefficients in an ILP model can be any real number.

We can expand ILP-based optimization problems to consider the weighted sum of multiple objectives as a minimization function, i.e.,

$$\min_{y} \sum_{k=1}^{K} w_k F_k(y) \quad \text{subject to} \quad \mathbb{B}\, y \leqslant c \tag{4.2}$$

where K is the total number of objectives, $F_k(y)$ is the k^{th} linear objective function, and w_k is the weight of $F_k(y)$. The optimization problem in Equation 4.2 is still a single-objective optimization problem, as it aims to minimize the value of one objective function.

A binary ILP problem is a special case of an ILP problem where the decision variables are restricted to binary values, i.e., $y \in \{0, 1\}^d$, and the rest of the ILP formulation looks the same in this case. In this thesis, we propose DCPO in Chapter 9—an ILP optimization problem where we want to find optimal places within the code to place DMA transfer calls. The decision variable in this problem takes the binary decision to place a DMA transfer call after a basic block or not. To solve the ILP-based optimizations, we use the `Gurobi` solver [Gur24] integrated within WCC.

4.3 Multi-Objective Optimization

Multi-objective optimizations are optimization problems that try to either minimize or maximize multiple objectives simultaneously. All the optimization problems we propose in this thesis aim to minimize objectives. So, for the sake of brevity, when we say optimize, we are referring to the minimization of objectives.

All approaches used to solve multi-objective optimization problems can be used to solve single-objective problems, but vice versa is not true. Single-objective optimization methods are not equipped to handle trade-offs between conflicting objectives. Multi-objective optimizations have multiple objectives

[1]The notation $\forall i = \overline{1, d}$ is analogous to $\forall i \in \{1, 2, \ldots, d\}$. However, it is not equivalent to $\forall i \in [1, d]$, as the interval $[1, d]$ also includes all real numbers between 1 and d.

that can contradict each other. Therefore, solutions to multi-objective problems are often a set of trade-off solutions known as Pareto optimal solutions.

Typically, heuristic approaches aim to optimize a single objective or the weighted sum of objectives that might not capture the complexity of the trade-offs between multi-objective solutions. They are fast, simple, and might converge to a solution, but they might get stuck in the local optimal region, missing the global picture. ILPs, on the other hand, are deterministic and guarantee the solution's optimality for the given ILP model, but they can be computationally expensive. Due to their deterministic nature, they follow a fixed path to find the optimal solution and potentially miss out on exploring other feasible regions of the solution space in the context of multi-objective optimization. Moreover, traditional ILP solvers are designed to find a single optimal solution, and it is challenging to find a Pareto optimal solution set using them. Even though metaheuristic algorithms do not guarantee optimality, they overcome all the shortcomings presented by heuristics and ILPs. Therefore, in this thesis, we chose to solve multi-objective optimizations using metaheuristic algorithms.

4.3.1 Problem Formulation

The following is the mathematical formulation for the multi-objective optimization problem.

$$\begin{aligned} \min_{x} \quad & F(x) = (F_1(x), F_2(x), \ldots, F_K(x)) \\ \text{subject to} \quad & Q_c(x) \leqslant 0, \text{ where } c = \overline{1, C} \end{aligned} \tag{4.3}$$

where $x \in X$ is the decision variable vector or solution vector and $X \subset \mathbb{R}^d$ is the decision or search space. $Y \subset \mathbb{R}^K$ is the objective space of the problem, $F : X \longmapsto Y$ is the objective function, and K is the total number of objectives. $Q_c(x)$ is the constraint applied to the optimization problem, and C is the total number of constraints applied to the problem.

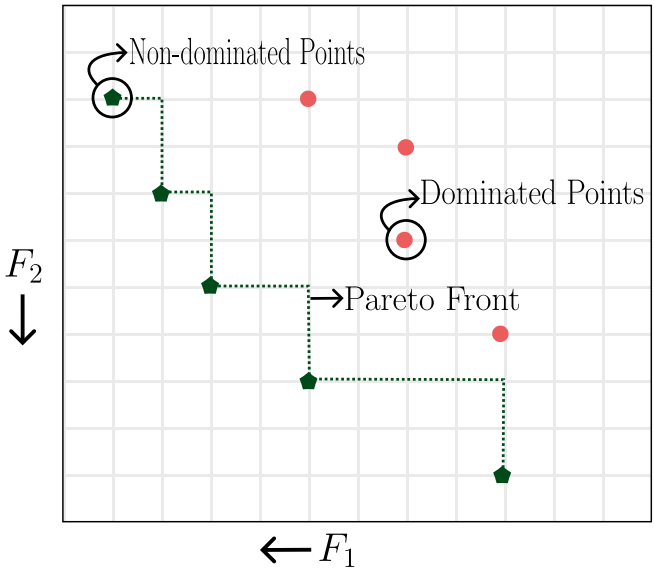
The optimization problem described in Equation 4.3 is a minimization problem. In the case of an optimization problem where the objective functions need to be maximized, we can formulate the problem by negating the objective functions that are being minimized or by using a maximization function.

4.3.2 Pareto Optimality

Unlike single-objective optimization, the optimality of solutions for multi-objective optimization is not straightforward. Therefore, the concept of Pareto optimality is used to evaluate the quality of solutions for such multi-objective optimization problems [Zit99, ED18, Ehr05].

Definition 4.1 (Pareto Dominance)
Let $x_1, x_2 \in X$ *be any two solution vectors, and* $F_k(x_1)$ *and* $F_k(x_2)$ *are objective*

Figure 4.1.: Exemplary Pareto front for a minimization problem. Let F_1 and F_2 be two objective functions that can contradict each other. The objective space for the minimization problem is 2-dimensional, and the objective values for the solutions are plotted in this objective space. The non-dominated and dominated points in the objective space are represented by green pentagons and red dots, respectively. The Pareto front, represented by a green dotted line, is formed by joining all non-dominated points.



values of x_1 *and* x_2*, where* $k = \overline{1,K}$ *and* K *is the total number of objectives. Then, for a minimization problem,* $x_1 \prec x_2$*, i.e.,* x_1 *dominates* x_2*, if* $\forall k = \overline{1,K}\ F_k(x_1) \leqslant F_k(x_2)$*, and there is at least one objective* $F_k(.)$ *where* $F_k(x_1) < F_k(x_2)$ *holds. For a maximization problem, the inverse inequations will hold true.*

Definition 4.2 (Weak Pareto Dominance)
Let $x_1, x_2 \in X$ *be any two solution vectors, and* $F_k(x_1)$ *and* $F_k(x_2)$ *are objective values of* x_1 *and* x_2*, where* $k = \overline{1,K}$ *and* K *is the total number of objectives. Then, for a minimization problem,* $x_1 \preceq x_2$*, i.e.,* x_1 *weakly dominates* x_2*, if* $\forall k = \overline{1,K}$ $F_k(x_1) \leqslant F_k(x_2)$*. For a maximization problem, the inverse inequations will hold true.*

As the names suggest, Pareto dominance is stronger than weak Pareto dominance. Any non-dominated solution is weakly non-dominated, but vice versa does not hold true.

Figure 4.1 shows multiple solutions in a 2-dimensional objective space for a minimization problem. All solutions in green (pentagon) have smaller F_1 and F_2 objective values than all solutions in red (dot). According to Definition 4.1, green pentagon solutions dominate red dot solutions.

Definition 4.3 (Pareto Optimal Solutions)
A solution vector $x \in X$ *is a Pareto optimal solution, if it is not dominated by any other solution.*

Definition 4.4 (Pareto Optimal Set)
The set P_s *of all Pareto optimal solutions is called the Pareto optimal set, i.e.,*

$$P_s = \{x \in X : x \textit{ is a Pareto optimal solution}\} \tag{4.4}$$

Definition 4.5 (Pareto Front)
The objective vectors corresponding to solutions in the Pareto optimal set form the Pareto front, i.e.,

$$PF = \{F(x) : x \in P_s\} \tag{4.5}$$

In Figure 4.1, if we compare solutions in green (pentagon) with each other, they have indifferent objectives, and we cannot state their dominance relations, i.e., they are not dominated by any other solution. Therefore, the solutions represented by green non-dominated points in the objective space are Pareto optimal solutions, and the set of all these solutions will be the Pareto optimal set. The green non-dominated points in Figure 4.1 form the Pareto front indicated by the green dotted line. All the solutions (red dots) that are above the green Pareto front are dominated by the solutions on the Pareto front (green pentagons) for the minimization problem.

Obtaining true Pareto fronts to multi-objective optimization problems proposed in this thesis is ambitious for the following reasons.

- The search space of the optimization problem depends on the underlying program, and it can often be large. Determining Pareto optimal solutions in a large search space might be NP-hard.
- Verifying the optimality of obtained solutions can be computationally infeasible.
- No general algorithm or function exists that can determine the program's WCET. Thus, static analysis techniques are used to safely estimate upper bounds for WCETs. Similarly, energy consumption is determined through experimental measurements and model building. Therefore, the objective values themselves can be close estimations of the actual WCET and energy consumption of the program.
- Due to the analytic nature of the objectives, we need to evaluate the objectives analytically using respective analyzers for every possible version of the source code to determine the true Pareto optimal objective values, which might be again NP-hard.

Knowles et al. [KTZ06] noted that for many real-world complex optimization problems, their search spaces can be quite large, making it NP-hard to identify

a single Pareto optimal solution. As a result, proving the optimality of such solutions may be infeasible or extremely computationally demanding. Therefore, a multi-objective optimization only aims to identify an approximated Pareto front that closely resembles the true Pareto front.

Consequently, under a realistic assumption that the true Pareto front is unknown, we define a new set PF_T that represents the best available approximation of the true Pareto front. For brevity's sake, when we talk about a Pareto front in this thesis, we are referencing such an approximation of the Pareto front. Let A and B be two Pareto fronts. The new set PF_T is defined as the set of all non-dominated points of the union of the sets A and B, i.e.,

$$PF_T = \{p_i | \forall p_i \nexists c_j \in (A \cup B) \prec p_i\} \tag{4.6}$$

In each chapter, during evaluations, we obtain the true Pareto front (PF_T) from the union of all the approximated Pareto optimal solutions obtained from all the evaluations of the respective approaches.

4.3.3 Quality Indicators

Just obtaining Pareto optimal solutions does not give a complete picture of the efficacy of the underlying optimization approach. To compare different optimization approaches and evaluate the quality of their Pareto front, we make use of four quality indicators in this thesis—Coverage, Non-dominance ratio, Ratio of non-dominated solutions, and Hypervolume.

Definition 4.6 (Coverage)
Coverage ($\mathcal{C}$) is a quality indicator that describes the total number of dominated points in a set A [Zit99].

$$\mathcal{C} = 1 - \frac{|\{a \in A : \exists p \in PF_T, a \preceq p\}|}{|A|} \tag{4.7}$$

If $\mathcal{C} = 1$, all the elements of PF_T dominate the elements of A. If $\mathcal{C} = 0$, all the elements in A dominate (or are equal to) the elements of the set PF_T. The lower the value of $\mathcal{C}$, the better.

Definition 4.7 (Non-Dominance Ratio)
Non-Dominance Ratio (NR) is a quality indicator that measures the ratio of non-dominated solutions contributed by a particular solution set A to the non-dominated solutions provided by all solution sets [GT08].

$$NR = \frac{|PF_T \cap A|}{|PF_T|} \tag{4.8}$$

The higher the number of solutions A contributes to the Pareto front PF_T, the higher the NR metric. The higher the value of NR, the better.

Definition 4.8 (Ratio of Non-Dominated Solutions)
*The Ratio of Non-dominated Solutions (*RNS*) is the quality indicator that calculates the number of non-dominated solutions concerning set A itself when compared to PF_T [TLK02].*

$$\texttt{RNS} = \frac{|a \in A : a \in PF_T|}{|A|} \tag{4.9}$$

The higher the number of non-dominated solutions on A, the higher the RNS metric. The higher the value of RNS, the better.

Definition 4.9 (Hypervolume)
*Hypervolume (*HV*) is the quality indicator that measures the volume of the objective space dominated by the non-dominated solutions with respect to an appropriately chosen reference point (*$\mathbf{R}$*), i.e.,*

$$\texttt{HV} = \lambda(\mathbb{D}(PF_F)) \tag{4.10}$$

where $\lambda(.)$ represents the Lbesgue measure (the volume) in $\mathbb{R}^q$, q is the dimension of the objective space, and $\mathbb{D}(PF_F) = \bigcup_{a \in PF_F} \{a' \in \mathbb{R}^q : a \prec a' \prec \mathbf{R}\}$ is the dominated region of the Pareto front PF_F with respect to $\mathbf{R}$ [Zit99, KFP+16].

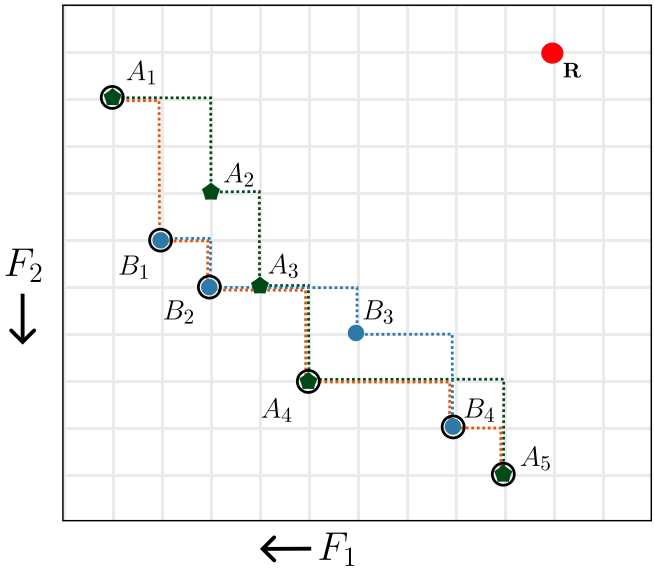
For a minimization problem, $\mathbf{R}$ has objective coordinates greater than all the solutions on the Pareto front. For the 2-dimensional objective space ($q = 2$) case, HV calculates the area of the region bounded by the reference point and the Pareto front, whereas HV calculates the volume in the 3-dimensional objective space ($q = 3$) case. The higher the area/volume enclosed by the Pareto front, the better the HV metric.

In Figure 4.2, we can see two approximated Pareto fronts, A and B, for an exemplary minimization problem with two objectives. The Pareto front A consists of 5 solutions indicated by green pentagons in the objective space, and B consists of 4 solutions indicated by blue circles in the objective space in the figure. The solutions on the true Pareto front PF_T obtained using A and B are demonstrated by black circles around the solutions in the figure, i.e., $PF_T = \{A_1, B_1, B_2, A_4, B_4, A_5\}$, and the Pareto front formed due to these solutions is represented using orange dotted lines.

Solution B_1 is better than A_2 in terms of both objectives; B_2 is equal to A_2 in terms of F_1 but better in terms of F_2; B_2 is also equal to A_3 in terms of F_2 but better in terms of F_1; A_4 is better than B_3 in terms of both objectives. Therefore, solutions A_2, A_3, and B_3 are not part of the true Pareto front PF_T.

$\mathbf{R}$ is the reference point used to calculate HV metrics for Pareto fronts A and B. To calculate the HV metric, we assume that each block of the grid in Figure 4.2 represents one unit value of objectives F_1 and F_2.

Figure 4.2.: Exemplary Pareto fronts to explain quality indicators for a minimization problem, where F_1 and F_2 are two contradicting objectives



From Figure 4.2 and the overall information about Pareto fronts A, B, and PF_T, we do the following calculations and get the following values for the quality indicators.

- The coverage value of A: $\mathcal{C}_A = (1 - 3/5) = 0.4$
- The coverage value of B: $\mathcal{C}_B = (1 - 3/4) = 0.25$
- The NR value of A : $\mathtt{NR}_A = 3/6 = 0.5$
- The NR value of B : $\mathtt{NR}_B = 3/6 = 0.5$
- The RNS value of A : $\mathtt{RNS}_A = 3/5 = 0.6$
- The RNS value of B : $\mathtt{RNS}_B = 3/4 = 0.75$
- The HV value of A : $\mathtt{HV}_A = (9 + 7 * 3 + 6 * 5 + 5 * 7 + 9) = 104$
- The HV value of B : $\mathtt{HV}_B = (8 * 4 + 7 * 5 + 6 * 4 + 8 * 2) = 107$

Unlike $\mathcal{C}$, NR, and RNS, the HV metric is not normalized between 0 and 1. Therefore, in this thesis, we also normalize the HV metric between 0 and 1 for consistency purposes. Consequently, we divide each HV metric with the total of all HV metrics, i.e., $\mathtt{HV}_A = 104/(104 + 107) = 0.493$ and $\mathtt{HV}_B = 107/(104 + 107) = 0.507$.

From these calculations, we can say that $\mathcal{C}_B$ is better than $\mathcal{C}_A$, $\mathtt{NR}_A$ and $\mathtt{NR}_B$ are nondominated, $\mathtt{RNS}_B$ is better than $\mathtt{RNS}_A$, and $\mathtt{HV}_B$ is better than $\mathtt{HV}_A$. Even in such a simple case, it is difficult to say which Pareto front is better. Pareto

front B has better $\mathcal{C}$, RNS, and HV metrics. However, if two Pareto fronts have the same value of a quality indicator, then the two Pareto fronts are nondominated in terms of that quality indicator. Pareto fronts A and B are nondominated in terms of NR, as shown in the case of NR_A and NR_B.

We can encounter cases where both Pareto fronts A and B contribute the same number of solutions to PF_T, and we get all quality indicator values are the same for A and B. In such cases, the quality of Pareto front A is indifferent to the quality of B, and we cannot decide on the better Pareto front.

4.4 Metaheuristic Algorithms

Metaheuristic algorithms can be used to solve multi-objective optimization problems. Unlike ILP-based approaches, metaheuristic algorithms do not guarantee optimality. However, they are more suitable for optimizing multiple objectives simultaneously. Moreover, unlike heuristic approaches, metaheuristic algorithms are less likely to get trapped in local optimal regions. Metaheuristic algorithms can be broadly classified in various categories, like nature-inspired, evolution-based, swarm intelligence-based, trajectory-based, etc. In this thesis, we use two metaheuristic algorithms: the Flower Pollination Algorithm (FPA) and the Strength Pareto Evolutionary Algorithm (SPEA).

However, there might exist better metaheuristic algorithms that might perform better than FPA and SPEA for the optimization problems proposed in this thesis. The choice of metaheuristic algorithms to solve a particular optimization problem is another big problem that we do not focus on in this thesis. It has been shown that FPA can outperform VEGA, NSGA-II, MODE, etc. [YKH14], and SPEA can outperform NSGA, VEGA, etc. [ZT99]. Therefore, we chose to solve the optimization problems proposed in this thesis using FPA and SPEA.

In this thesis, we use the following terminology in the context of metaheuristic algorithms:

- **Individual:** The pair of a solution or decision vector x and its associated objective vector $F(x)$, i.e., $(x, F(x))$, is called an Individual.
- **Population:** The set of individuals is called a Population. The size of the population is often predefined.
- **Generation:** Metaheuristic algorithms are iterative in nature. Each iteration, where the individuals within the population are updated, is called a Generation. The maximum number of generations is often predefined.
- **Stopping Criteria:** If the optimal solutions are unknown, then we need to predefine stopping criteria to halt the algorithm's iteration. The following are some of the stopping criteria we can use: maximum

Algorithm 4.1 Generic metaheuristic algorithm

1: **Input:** Initial population
2: **Output:** Pareto front
3: **while** Stopping criteria are not reached **do**
4: **for** each Individual in the current Generation **do**
5: Update Individual using a **local** or **global** update operator or both
6: Select Individuals for the next Generation using the **selection** operator

number of generations, maximum number of generations without change in population, maximum CPU time, etc.

- **Initial Population:** To start the iterative metaheuristic algorithm run, we need to generate an initial set of individuals that serves as a starting point for the algorithm. The initial population could be generated using problem-specific details, but in this thesis, we randomly generate the initial population (unless specified otherwise). As metaheuristic algorithms do not guarantee optimality, the initial population influences the convergence of the algorithm. Therefore, we solve the optimization problem multiple times using different initial populations to reduce the influence of randomness on the final results.

- **Update Operator:** To update an individual, every metaheuristic algorithm uses different strategies. The update operators are broadly categorized as local and global operators. Local operators combine information from existing individuals from the current population to generate a new individual; global operators are used to update individuals so that the design space exploration does not get stuck in some local region.

- **Selection Operator:** Before jumping to the next generation, the population is updated using newly generated individuals. The metaheuristic algorithm employs a selection operator to select top-scoring individuals for the next generation. In this thesis, we use a selection operator based on Pareto dominance [ED18] to collect the top-scoring individuals and pass them to the next generation.

Algorithm 4.1 describes a generic metaheuristic algorithm. An initial population of a fixed size serves as an input to the algorithm, and an approximated Pareto optimal front is the output (Lines 1 and 2). We iterate through generations as long as predefined stopping criteria are not reached (Line 3). During each generation, each individual is updated using either global or local or both update operators (Lines 4 and 5). The updated individuals are evaluated to get their objective values, and using the selection operator, the individuals for the next generation are selected (Line 6).

While using metaheuristic algorithms to solve an optimization problem, we maintain an archive of all evaluated individuals and their respective objective values. If the metaheuristic algorithm encounters a previously evaluated individual, we skip its evaluation and instead use the previously cached objective values from the archive.

After reaching the stopping criteria, the metaheuristic algorithm selects Pareto optimal solutions out of the final generations. Every metaheuristic algorithm has diverse update and selection operators, all aiming to optimize the problem's multiple objectives. In Subsections 4.4.1 and 4.4.2, we briefly describe two such metaheuristic algorithms—SPEA and FPA—used to solve multi-objective optimization problems in this thesis.

4.4.1 Strength Pareto Evolutionary Algorithm (SPEA)

The Strength Pareto Evolutionary Algorithm (SPEA) is an evolutionary algorithm introduced by Zitzler [Zit99]. SPEA is a population-based evolutionary algorithm—a subcategory of metaheuristic algorithms—that uses selection, recombination, and mutation techniques iteratively to find Pareto optimal solutions.

SPEA maintains a population $\mathbf{P}^g$ and an external set $\bar{\mathbf{P}}^g$, where $g = \overline{1, maxGen}$, and *maxGen* is the maximum number of generations for which SPEA iterates. The population $\mathbf{P}^g$ has N individuals, each denoted by $x^{(g,n)} \in \mathbf{P}^g$, $n = \overline{1, N}$, and the external set $\bar{\mathbf{P}}^g$ contains $\bar{N}$ non-dominated individuals $x^{(g,\bar{n})} \in \bar{\mathbf{P}}^g$, $\bar{n} = \overline{1, \bar{N}}$.

At generation g, $\bar{\mathbf{P}}^{g+1}$ for the next generation is computed by adding together all individuals from $\bar{\mathbf{P}}^g$ and non-dominated individuals from $\mathbf{P}^g$. Then, all weakly dominated solutions are consecutively removed from $\bar{\mathbf{P}}^{g+1}$. Finally, a clustering algorithm is applied to limit the size of the external set $\bar{\mathbf{P}}^g$ to $\bar{N}$. Furthermore, the fitness values of individuals in $\mathbf{P}^g$ and $\bar{\mathbf{P}}^g$ are computed, which is the basis for the selection phase.

The fitness assignment is the core of this algorithm. It uses a notion of *strength* to assign a fitness value to each individual. First, each individual from the external set $\bar{\mathbf{P}}^g$, i.e., $x^{(g,\bar{n})} \in \bar{\mathbf{P}}^g$, $\bar{n} = \overline{1, \bar{N}}$, is assigned a strength value $S(x^{(g,\bar{n})})$ using the following equation.

$$\forall x^{(g,\bar{n})} \in \bar{\mathbf{P}}^g : S(x^{(g,\bar{n})}) = \frac{\left|\{x^{(g,n)} | x^{(g,n)} \in \mathbf{P}^g \wedge x^{(g,\bar{n})} \preceq x^{(g,n)}\}\right|}{N+1} \tag{4.11}$$

The strength value $S(x^{(g,\bar{n})}) \in [0, 1)$ is a real value, which is proportional to the number of population individuals $x^{(g,n)} \in \mathbf{P}^g$, for which $x^{(g,\bar{n})}$ weakly dominates $x^{(g,n)}$ ($x^{(g,\bar{n})} \preceq x^{(g,n)}$). These strength values, calculated using the external set $\bar{\mathbf{P}}^g$, are used to calculate the fitness values for the individuals in $\mathbf{P}^g$. In the second step, the fitness value is assigned to each $x^{(g,n)} \in \mathbf{P}^g$. The

fitness is given by the accumulated strength of all individuals of the external set $\bar{\mathbf{P}}^g$ that weakly dominate the individual of the population $\mathbf{P}^g$:

$$\forall x^{(g,n)} \in \mathbf{P}^g : \mathcal{F}(x^{(g,n)}) = 1 + \sum_{\forall x^{(g,\bar{n})} \in \bar{\mathbf{P}}^g : x^{(g,\bar{n})} \preceq x^{(g,n)}} \mathcal{S}(x^{(g,\bar{n})}) \tag{4.12}$$

where $\mathcal{F}(x^{(g,n)}) \in [1, N)$. SPEA uses an external set to store nondominated individuals from the population. It assigns scalar fitness values to individuals from the population at each generation using Equations 4.11 and 4.12. Once the fitness values are assigned to the individuals, the selection is performed.

During the selection phase, a mating pool $\tilde{\mathbf{P}}^g \subset \mathbf{P}^g + \bar{\mathbf{P}}^g$ is used to store the individuals. Two individuals $x^{(g,\tilde{N}_1)}, x^{(g,\tilde{N}_2)} \in \mathbf{P}^g + \bar{\mathbf{P}}^g$ are chosen randomly, and the individual with the lower fitness value is added to the mating population $\tilde{\mathbf{P}}^g$, i.e., *If* $\mathcal{F}(x^{(g,\tilde{N}_1)}) < \mathcal{F}(x^{(g,\tilde{N}_2)})$, *then* $\tilde{\mathbf{P}}^g = \tilde{\mathbf{P}}^g + \{x^{(g,\tilde{N}_1)}\}$ *else* $\tilde{\mathbf{P}}^g = \tilde{\mathbf{P}}^g + \{x^{(g,\tilde{N}_2)}\}$. This step is repeated N times to fill the mating pool for the recombination phase.

The recombination process is similar to the selection process, where we choose two random individuals $x^{(g,\tilde{N}_1)}, x^{(g,\tilde{N}_2)} \in \tilde{\mathbf{P}}^g$, and create a new individual with single-point crossover by using features either from $x^{(g,\tilde{N}_1)}$ or $x^{(g,\tilde{N}_2)}$, with the crossover probability p_c. Using two d-dimensional parent individuals, a single-point crossover takes a union of the front d' entries of the first parent individual and the rear $(d - d')$ entries of the second parent individual to create a new child individual, where $d' = \overline{0, d-1}$ is chosen randomly. The recombination in SPEA refers to the local operator of Algorithm 4.1.

Furthermore, mutation, which describes random changes to an individual's features with a probability of p_m, can also be performed. For example, if a bit vector represents an individual, then each bit can flip with a probability of p_m. We use a multi-bit mutation strategy that mutates multiple bits in an individual randomly to create a new mutated individual. The mutation in SPEA refers to the global operator of Algorithm 4.1. The algorithm stops when a stopping criterion is fulfilled, and the external set $\bar{\mathbf{P}}^g$ is presented as an output.

During evaluations, we need to set the following hyperparameters before solving an optimization problem using SPEA:

- Crossover probability;
- Mutation probability;
- Size of the population;
- Size of the external set;
- Maximum number of generations.

4.4.2 Flower Pollination Algorithm (FPA)

A Flower Pollination Algorithm (FPA) [Yan12, YKH14] is inspired by the flower pollination process seen in the flowering of plants. FPA maintains a

population set $\mathbf{P}^g$ with N individuals. FPA uses local and global pollination operators to transform individuals from $\mathbf{P}^g$ at every generation g. A switch probability $p_s \in [0, 1]$ is defined to switch between the local and the global pollination process in FPA. The switch probability is slightly biased towards local pollination, as the probability of local pollination occurring in nature is higher than the occurrence of global pollination.

Within FPA, we use a pollination operator Φ to update the current individual. The pollination operator $\Phi : X \longmapsto \mathbb{R}^d$ maps the current individual $x^{(g,n)} \in X$ to $\Phi(x^{(g+1,n)}) \in \mathbb{R}^d$. Depending on the switch probability, the pollination operator Φ can be represented by either Equation 4.13 or Equation 4.17.

To perform global pollination for an individual $x^{(g,n)} \in \mathbf{P}^g$, we use the following global pollination operator within FPA.

$$\Phi(x^{(g+1,n)}) = x^{(g,n)} + \gamma L(\lambda)(\mathbf{BI}^g - x^{(g,n)}) \tag{4.13}$$

where $\mathbf{BI}^g$ represents the best individual from generation g, and γ is a scaling factor to control the step size while updating to the next generation. $L(\lambda)$ corresponds to the Lévy flights' [Pav07] based step size, which helps to mimic the characteristics of the pollinator's flight pattern over long distances to perform global pollination. In Equation 4.13, $L(\lambda)$ is a large pseudo-random step size, which helps the current individual $x^{(g,n)}$ to escape the local landscape and update to $\Phi(x^{(g+1,n)})$ by using the current best individual $\mathbf{BI}^g$. $L > 0$ is drawn from a Lévy distribution:

$$L \sim \frac{\gamma\Gamma(\lambda)\sin(\pi\lambda/2)}{\pi}\frac{1}{s^{1+\lambda}}, \quad (s \gg s_0 > 0), \tag{4.14}$$

where $\Gamma(\lambda)$ is a standard gamma function, λ is a positive integer, and the above-mentioned Lévy distribution is valid for large steps $s > 0$. One of the ways to draw such random numbers is the so-called Mantegna algorithm [Man94] for drawing step size s by using two Gaussian distributions.

In a multi-objective optimization problem, we can have individuals with objectives that could contradict each other. Therefore, finding the best individuals $\mathbf{BI}^g$ from generation g becomes a difficult task. One of the approaches is to take the weighted sum of multiple objectives and combine them into a single composite objective. The single composite objective F_c is defined as follows:

$$F_c = \sum_{k=1}^{K} w_k F_k, \text{ where } \sum_{k=1}^{K} w_k = 1, \text{ and } w_k > 0 \tag{4.15}$$

where F_k represents the k^{th} objective function, and w_k are non-negative weights. To obtain a good approximation of the Pareto front, which has solutions relatively uniformly distributed on it, we should use random weights that are as different as possible [Mie12, YKH14]. Therefore, in practice, each

objective function F_k of a particular solution is assigned a random weight at each iteration. By assigning pseudo-random weights to the objective functions during each iteration, we provide equally distributed importance to all objectives and aim to obtain a broad approximation of the Pareto front. To get an approximation of the Pareto front with individuals uniformly distributed on the Pareto front, we draw random weights w_k using the following equation.

$$w_k = \frac{v_k}{\sum_{k=1}^{K} v_k} \tag{4.16}$$

where v_k are random numbers drawn from a uniform distribution $[0, 1]$. By dividing v_k with $\sum_{k=1}^{K} v_k$, we are normalizing w_k so that the condition $\sum_{k=1}^{K} w_k = 1$ is satisfied. After calculating a composite single objective for each individual from the population, we can find the best individual $\mathbf{BI}^g$. The best individual $\mathbf{BI}^g$ is only employed to update an individual during global pollination in Equation 4.13.

To perform local pollination for an individual $x^{(g,n)} \in \mathbf{P}^g$, we use the following local pollination operator within FPA.

$$\Phi(x^{(g+1,n)}) = x^{(g,n)} + \epsilon(x^{(g,N_1)} - x^{(g,N_2)}) \tag{4.17}$$

where $x^{(g,N_1)}$ and $x^{(g,N_2)}$ are the individuals selected from the local neighborhood of $x^{(g,n)}$. In Equation 4.17, ϵ is drawn from a uniform distribution between the interval $[0, 1]$, which helps to update the current individual $x^{(g,n)}$ to $\Phi(x^{(g+1,n)})$ comprising characteristics from its local neighborhood. This update equation mimics the behavior of a local random walk.

After performing either global or local pollination, the individuals are updated in a continuous-valued search space. However, in this thesis, we deal with binary decision vectors. Therefore, after updating to the next generation, we restrict the new individuals to binary-valued vectors only. For the vector $\Phi(x^{(g+1,n)}) \in \mathbb{R}^d$, operator $\mathbb{L}$, which is a standard logistic function, is defined as follows:

$$\mathbb{L}(\Phi(x_i^{(g+1,n)})) = \frac{1}{1 + \exp^{-\Phi(x_i^{(g+1,n)})}} \tag{4.18}$$

where $\mathbb{L}(\Phi(x_i^{(g+1,n)})) \in (0, 1)$ is calculated for each decision variable of the vector $\Phi(x^{(g+1,n)}) \in \mathbb{R}^d$. Then, a new binary individual vector $x^{(g+1,n)}$ is generated using Equation 4.19:

$$x_i^{(g+1,n)} = \begin{cases} 1, \text{ if } \mathbb{L}(\Phi(x_i^{(g+1,n)})) > \mu, \\ 0, \text{ otherwise} \end{cases} \tag{4.19}$$

where $\mu \sim U(0,1)$ is the sample drawn from a uniform distribution between the interval $[0,1]$, and $x_i^{(g+1,n)}$ is the i^{th} decision variable in the individual vector $x^{(g+1,n)}$.

Once all the individuals in the g^{th} generation are updated and evaluated, the selection phase is employed. During the selection phase, which is responsible for collecting the top-scoring individuals and passing them to the next $(g+1)^{th}$ generation, FPA uses the concept of Pareto dominance (cf. Definition 4.1). Therefore, during the selection phase, we compare all the solutions from the current population $\mathbf{P}^g$ and the updated individuals, collect N best-scoring individuals based on Pareto dominance, and pass them to the next generation's population $\mathbf{P}^{g+1}$. The algorithm stops when the stopping criteria are fulfilled, and the final approximated Pareto front is presented as an output.

During evaluations, we need to set the following hyperparameters before solving an optimization problem using FPA:

- Switch probability;
- Scaling factor;
- λ for standard gamma function;
- Size of the population;
- Maximum number of generations.

In the upcoming chapters, we use both SPEA and FPA algorithms to solve multi-objective optimization problems. Section 5.3 describes the integration of metaheuristic algorithms within the compiler-level multi-objective optimization framework in the context of static SPM allocation. Section 5.4.1 describes the hyperparameters and the evaluation setup used to solve the optimizations using these metaheuristic algorithms.

Part II

Static Allocation

5

Multi-Objective Static SPM Allocation

In this chapter, we propose a compiler-level static SPM allocation-based multi-objective optimization to minimize the program's WCET and energy consumption simultaneously. As mentioned in Subsection 2.4.2.3, SPMs are small but fast and energy-efficient local memories compared to the Flash. The performance and energy efficiency of SPMs can be leveraged to optimize the code at the compiler level.

Static SPM allocation is one such memory allocation-based compiler optimization. We can use the static SPM allocation to move assembly-level program objects (for example, basic blocks and global data variables) from the slow Flash memory to the fast SPM memory. These objects are executed faster and consume less energy when they are executed from the SPM compared to when they are executed from the Flash. After performing static SPM allocation, the objects are loaded in the specified memory only once during startup and executed from that memory during runtime.

However, as SPMs are small in size, we cannot fit the whole program in the SPM in most cases. Therefore, the main crux of the optimization problem arises: which object must be allocated to which memory to get optimal results?

In the past, single-objective SPM allocation-based optimizations have been explored to optimize a program's WCET or energy consumption. However, WCET and energy objectives can contradict each other, i.e., minimizing energy consumption can increase a program's WCET and vice versa. If we use a single-objective SPM allocation-based optimization to minimize WCET, it might adversely affect the program's energy consumption and vice versa. Therefore, this chapter considers multiple contradicting objectives like WCET and energy consumption and tries to minimize them simultaneously using multi-objective optimization. This chapter aims to:

- identify if a multi-objective static SPM allocation-based optimization can provide better solutions than the state-of-the-art single-objective ILP-based static SPM allocation.

- understand if the multi-objective static SPM allocation can provide multiple Pareto optimal solutions compared to the single solution provided by the single-objective ILP-based static SPM allocation.

- identify the bottlenecks while performing the proposed multi-objective static SPM allocation-based optimization.

Consequently, in this chapter, we construct a static SPM allocation-based multi-objective optimization problem that considers multiple system properties simultaneously as minimization objectives, aims to find solutions that fulfill all criteria, and explores potential trade-offs between objectives. The multi-objective static allocation optimization problem defined in this chapter is for a single-task system. The proposed multi-objective optimization aims to find optimal program object allocations to minimize the program's WCET and energy consumption simultaneously. For brevity's sake, we hereafter refer to the multi-objective static SPM allocation optimization as MO_S.

We use two metaheuristic algorithms—SPEA and FPA—to solve the multi-objective MO_S problem (cf. Section 4.4). The FPA-based framework for the approach described in this chapter was presented at the International Workshop on Software and Compilers for Embedded Systems (SCOPES) in 2019 [JF19].

The MO_S problem consists of architecture-independent and architecture-dependent parts. The solutions, objectives, minimization problem formulation, and metaheuristic algorithms are architecture-independent. However, the code compiled by the compiler, the costs associated with code transformations, and objective analyses are architecture-dependent. By definition, it is impossible to formulate a complete architecture-independent low-level compiler optimization. The key contributions of this chapter are:

- We propose a compiler-level static SPM allocation-based multi-objective optimization (MO_S) problem that aims to minimize the program's WCET and energy consumption.

- We solve the multi-objective static SPM allocation optimization using two metaheuristic algorithms—FPA and SPEA.

During evaluations, we demonstrate the utility of the proposed MO_S optimization by compiling, analyzing, and optimizing code for the ARM Cortex-M0 processor using the WCC compiler. WCC is equipped with different state-of-the-art ILP-based single-objective optimizations [Fal09, FSS11, FK09, FL10]. We compare the performance and solution quality of the MO_S optimization with a WCET-minimizing version of the state-of-the-art ILP-based single-objective static SPM allocation optimization implemented within WCC. For brevity's sake, we refer to the ILP-based single-objective static SPM allocation optimization as SO_S. We also compare MO_S with a BaseLine (BL) evaluation, where the whole program is executed from the Flash memory. In this chapter, we explore the potential of the proposed multi-objective MO_S optimization compared to the state-of-the-art single-objective SO_S and BL. The key results from the evaluations are:

- We compare the proposed MO_S problem with single-objective ILP-based optimization (SO_S) and a baseline evaluation (BL).
- MO_S–FPA and MO_S–SPEA performed slightly better than SO_S; both MO_S and SO_S performed better than BL.
- MO_S–SPEA provided better solutions qualitatively and quantitatively compared to MO_S–FPA.

This chapter is organized as follows: Section 5.1 provides an overview of the related background work. Section 5.2 presents the problem formulation for the static SPM allocation-based multi-objective optimization. Section 5.3 presents the proposed metaheuristic algorithm-based MO_S optimization. Section 5.4 presents the evaluation results, and Section 5.5 concludes the chapter.

5.1 Related Work

Allocation problems, either of registers or memories, are among the most researched and fundamental topics in compiler construction and code optimizations. Classical register allocators initially assume the code uses an infinite number of pseudo-registers. Later, they perform allocation by mapping them to a limited number of the processor's real registers [Fal09]. Similarly, the memory allocation problem involves allocating specific parts of an application to the memory. Applications can either request memory or release some of the already allocated memory. The memory allocation problem needs to be addressed to fulfill these requests while maintaining their objectives in mind. Both register and memory allocation problems are proven to be NP-hard.

There are variations between the speeds and capacities at which different processors' memories operate. Therefore, the memory allocation problem can exploit these discrepancies while optimizing different objectives. As mentioned in Subsection 2.4.2.2, caches are small and fast memories between the processor and the Flash. However, due to the unpredictable runtime behavior of caches, techniques like cache locking for real-time systems can be employed to exploit caches at the compiler level.

Puaut et al. [PD02] presented two heuristics for cache content selection, i.e., to lock an instruction cache to minimize CPU utilization and interference between multiple tasks. Compoy et al. [CPI+05] proposed a genetic algorithm for cache locking. Falk et al. [FPT07] proposed an approach to minimize WCET by performing static instruction cache locking using worst-case path construction and instruction cache content selection. Plazar et al. [PKM+12] proposed an ILP-based static locking of instruction caches. In these approaches, the content placed within the caches was locked to have real-time predictability, functionally equating caches closely to SPM.

As mentioned before, SPM is an on-chip SRAM, which is much better in terms of speed and energy consumption than off-chip memories. Techniques

like static and dynamic SPM allocation can be employed to exploit SPMs at the compiler level. Memory allocation problems can be classified broadly into two categories: one that solely considers data allocation [SMRC05, BDC+07, KC02, SS08] and another that considers code allocation [FK09, AMF+04, WXP10]. Furthermore, some approaches considered the allocation of both data and code while solving a memory allocation problem [LZ15].

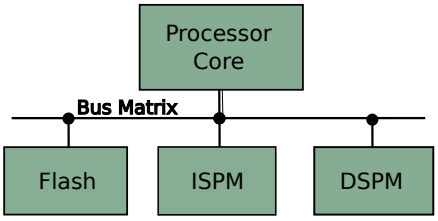
In this chapter, we focus on assembly-level static SPM allocation, which allocates code and data from the Flash to SPM at compile-time. The compiler-level static memory allocation approach loads program objects once into a specific memory during startup. Then, during the execution of the entire application, the content within the memory is not changed.

Suhendra et al. [SMRC05] presented static data SPM allocations using ILP and two different WCET-aware heuristics. Falk et al. [FK09] modified and extended this approach to perform ILP-based WCET-aware static SPM allocation for program code. A version of this ILP-based single-objective static SPM allocation was already implemented within WCC, referred to as SO_S in this thesis. Unlike the proposed MO_S, SO_S considers static allocation of only basic blocks—static allocation of global data objects is not considered. As SO_S aims to minimize WCET, we perform an energy analysis on the final solution obtained after solving SO_S to get the respective energy results. We compare the results of SO_S with the proposed MO_S during evaluations.

Steinke et al. [SWLM02] proposed static SPM allocation of functions, basic blocks, and data to minimize energy. Similarly, Srikant and Vardhan [SS08] presented a static SPM allocation for data objects to optimize energy consumption. Balasa et al. [BAG+17] proposed an energy-aware data allocation for systems with multiple SPMs. Ishitobi et al. [IIY07] proposed an algorithm to find optimal code placement for SPM and cacheable and non-cacheable regions. Balasundaram et al. [BC15] proposed a metaheuristics-based optimizer to find the optimal code layout that minimizes energy consumption. These approaches used either ILP-, heuristic-, or evolutionary-based techniques focusing on optimizing a single objective without considering its impact on the other code properties.

Multi-objective optimizations are rarely exploited to perform compiler-level optimization. Hoste et al. [HE08] proposed a framework that uses an evolutionary algorithm-based multi-objective optimization to explore various compiler optimization levels that automatically find Pareto optimal optimization levels. Lokuciejewski et al. [LPF+11] presented a stochastic evolutionary approach to find Pareto optimal compiler optimization sequences for a bi-objective problem. They considered trade-offs between ACET and WCET as well as WCET and code size. Balaprakash et al. [BTW13] proposed a multi-objective code optimization of high-performance computing kernels to explore potential trade-offs between performance, power, and energy. Muts et al. [MLF18] proposed a high-level function-inlining-based multi-criteria compiler optimization that used Generalized Differential Evolution (GDE3) and Multi-objective Binary

Figure 5.1.: An exemplary architecture for static SPM allocation



Probability Optimization Algorithm (MBPOA) to minimize WCET, energy consumption, and code size. However, to the best of our knowledge, none of these multi-objective optimizations focus on low-level compiler optimizations and static SPM allocation to exploit the memory subsystems.

5.2 Problem Formulation

Static SPM allocation is an assembly-level compiler optimization. It is used to allocate program objects from the Flash memory to the SPM statically at compile time. In this chapter, we consider static SPM allocation of basic blocks and global data variables (also referred to as program objects). We do not consider local data objects during static allocation as they are usually stored on the stack, and it would lead to heavy stack frame reorganization, which is beyond the scope of this thesis.

SPMs are faster and more energy efficient than the Flash. However, they are small and mostly cannot fit the complete program. Moreover, placing a basic block in SPM can break jumps between its predecessor and successor. So, we need to perform code transformations to fix broken jumps, which can increase the code size and incur extra costs in terms of WCET and energy consumption. Unlike basic blocks, static allocation of global data variables does not incur extra code transformations.

Figure 5.1 shows an exemplary architecture that is considered in this chapter for static SPM allocation. This underlying architecture consists of a Flash memory, an instruction SPM (ISPM), and a data SPM (DSPM). Henceforth, unless explicitly stated, an SPM always refers to the ISPM in this chapter. During evaluations, we generate statically allocated code, perform WCET and energy analyses, and solve MO_S and SO_S optimizations for ARM Cortex-M0. Figure 5.1 exactly represents the STM32F0xx implementation of the ARM Cortex-M0 architecture [STM22a].

In this section, we define a compiler-level multi-objective SPM allocation-based optimization that aims to simultaneously minimize the WCET and energy consumption of a program. The reduction in WCET and energy consumption can be achieved by optimally placing basic blocks in ISPM and global data variables in DSPM. The multi-objective SPM allocation problem can be mathematically formulated as a minimization problem as follows:

$$\begin{aligned} \min_{x} \quad & F(x) = (F_1(x), F_2(x)) \\ \text{subject to} \quad & Q_1(x_{1:\tilde{d}}) \leqslant 0 \text{ and } Q_2(x_{(\tilde{d}+1):d}) \leqslant 0 \end{aligned} \tag{5.1}$$

The function $F : X \longmapsto Y$ maps a decision vector from the search space $X \subset \{0,1\}^d$ to a 2-dimensional objective space $Y \subset \mathbb{R}^2$. $F_1(x)$ and $F_2(x)$ represent WCET and energy objectives of the optimization problem, respectively. The search space X consists of $x = (x_1, .., x_{\tilde{d}}, x_{(\tilde{d}+1)}, .., x_d) \in X$, a d-dimensional binary decision variable vector. The subvector $x_{1:\tilde{d}}$ represents the decision vector for basic blocks, where $\tilde{d}$ represents the total number of basic blocks in the program and the decision variable $x_i \in \{0,1\}\ \forall i = \overline{1, \tilde{d}}$ decides whether a basic block b_i is placed in ISPM or Flash, i.e.,

$$\forall i = \overline{1, \tilde{d}} \quad x_i = \begin{cases} 1, & \text{if } b_i \text{ is placed in ISPM,} \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

The subvector $x_{(\tilde{d}+1):d}$ represents the decision vector for the global data variables, where $(d - \tilde{d})$ is the total number of global data variables. The decision variable $x_g \in \{0,1\}\ \forall g = \overline{(\tilde{d}+1), d}$ decides whether a global data variable $\mathcal{G}_g$ is placed in DSPM or Flash, i.e.,

$$\forall g = \overline{(\tilde{d}+1), d} \quad x_g = \begin{cases} 1, & \text{if } \mathcal{G}_g \text{ is placed in DSPM,} \\ 0, & \text{otherwise} \end{cases} \tag{5.3}$$

The constraint condition $Q_1(x_{1:\tilde{d}}) \leqslant 0$ for the SPM allocation problem represents the number of basic blocks that can fit in the ISPM statically.

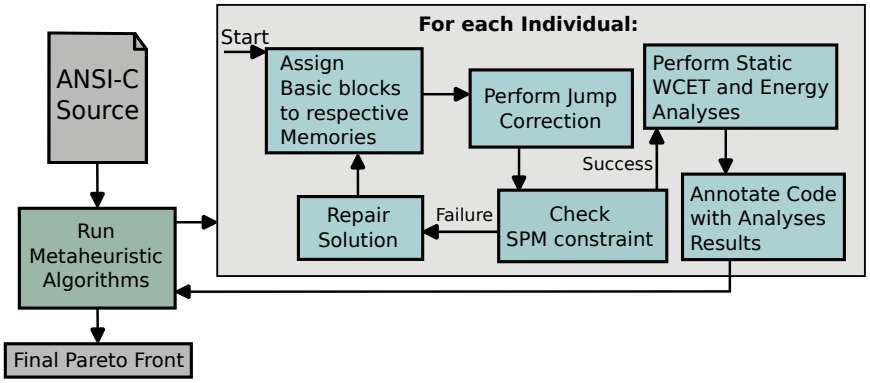
$$Q_1(x_{1:\tilde{d}}) = \sum_{i=1}^{\tilde{d}} B_i x_i + \Psi_J - S_{ISPM} \tag{5.4}$$

where B_i is the code size of basic block b_i, S_{ISPM} is the total size of the ISPM, and Ψ_J represents the extra code size cost incurred due to jump correction.

As mentioned in Subsection 3.1.3, performing SPM allocation can change the physical successors and predecessors of basic blocks, alter memory addresses, and invalidate previous jumps between them. To repair instructions, we need to add jump correction code and ensure the proper control flow of the program. Depending on the type of jump between basic blocks and the underlying architecture, the requirement for additional jump correction code may vary.

The term $(\sum_{i=1}^{\tilde{d}} B_i x_i + \Psi_J)$ in Equation 5.4 represents the total code size after assigning basic blocks to respective memories and performing jump correction. To check if the constraint condition $Q_1(x_{1:\tilde{d}}) \leqslant 0$ is satisfied, we simply assign basic blocks to either Flash or SPM according to the decision vector x, perform the necessary jump corrections, and then calculate the total size of the generated code within WCC.

Figure 5.2.: Multi-objective static SPM allocation optimization framework



A relatively precise model for the cost Ψ_J can be used to predict the extra code size cost incurred due to jump correction. In the case of ILP-based static allocation, we need to include the effects of jump correction and the extra costs incurred within the ILP model [Lup20]. However, while using metaheuristic algorithms, we can easily generate a jump-corrected code and precisely measure the code size without any errors. Therefore, we do not need to estimate the approximate value of Ψ_J in this thesis.

Finally, the constraint condition $Q_2(x_{(\tilde{d}+1):d}) \leqslant 0$ applied on the optimization problem (MO_S) proposed in Equation 5.1 represents the number of global data variables that can fit in the DSPM statically.

$$Q_2(x_{(\tilde{d}+1):d}) = \sum_{g=(\tilde{d}+1)}^{d} S_{\mathcal{G}_g} x_g - S_{DSPM} \tag{5.5}$$

where $S_{\mathcal{G}_g}$ is the size of the global data variable $\mathcal{G}_g$, and S_{DSPM} is the total size of the DSPM.

5.3 Multi-Objective Static SPM Allocation Optimization

In this section, we exploit the metaheuristic algorithms to solve the multi-objective static SPM allocation optimization (MO_S) problem proposed in Equation 5.1. Figure 5.2 shows the steps to perform multi-objective MO_S optimization and to generate statically allocated code within WCC. Moreover, Algorithm 5.1 describes the pseudo-code for solving the MO_S problem.

To solve MO_S, we first define the optimization problem, the search space dimension using the assembly-level basic block information, the objectives, and the SPM size constraint (Line 1). We use metaheuristic algorithms—FPA and SPEA—to solve the MO_S problem. We randomly initialize the initial population of metaheuristic algorithms (Line 3). We perform evaluations using P initial populations and run the metaheuristic algorithm to solve

Algorithm 5.1 Multi-Objective Static SPM Allocation (MO_S)

1: Define the MO_S problem (search space, objectives, and SPM size constraint)
2: **for** $p = 1 : P$ **do** ▷ Run algorithm for P different initial populations
3: Randomly initialize the initial population of size N
4: **for** $n = 1 : N$ **do** ▷ Iterate over each individual in the initial population
5: Assign basic blocks to ISPM and global data variables to DSPM according to the initial individual vector
6: **while** $Q_1(x_{1:\tilde{d}}) > 0$ **do** ▷ Check the ISPM size constraint
7: Repair the solution by removing a basic block from ISPM
8: **while** $Q_2(x_{(\tilde{d}+1):d}) > 0$ **do** ▷ Check the DSPM size constraint
9: Repair the solution by removing a data variable from DSPM
10: Perform jump correction for the initial individual
11: Perform WCET and energy analyses
12: Initialize hyperparameters for the metaheuristic algorithms
13: **while** Stopping criteria are not reached **do** ▷ Iterate until the stopping criteria are fulfilled
14: **for** $n = 1 : N$ **do** ▷ Iterate over each individual in the current population
15: Update the individual using a global or local operator or both
16: **while** $Q_1(x_{1:\tilde{d}}) > 0$ **do** ▷ Check the ISPM size constraint
17: Repair the solution by removing a basic block from ISPM
18: **while** $Q_2(x_{(\tilde{d}+1):d}) > 0$ **do** ▷ Check the DSPM size constraint
19: Repair the solution by removing a data variable from DSPM
20: Perform jump correction and evaluate the current individual
21: Update the next generation using the selection operator
22: Collect the Pareto optimal solutions obtained for p^{th} initial population
23: **Return:** Final Pareto optimal solution set (PF_F)

the MO_S problem P times for different initial populations (Line 2). As the obtained Pareto front is an approximation, the initial population influences the convergence of the metaheuristic algorithm. Therefore, we generate the final Pareto front (PF_F) using the Pareto fronts obtained from each of the P metaheuristic algorithm runs to reduce the influence of randomness on the final results (Lines 22 and 23) (cf. Section 4.4).

For each individual in the initial population, we assign the basic blocks to ISPM and global data variables to DSPM according to the individual decision vector (Line 5) and check if it satisfies the SPM size constraints (cf. Equations 5.4 and 5.5). If the ISPM overflows due to the current basic block assignment, we repair the solution by randomly assigning basic blocks to the Flash until the constraint is satisfied (Lines 6 and 7). Similarly, we repair the individual by randomly assigning global data variables back to the Flash if the DSPM is overflowing (Lines 8 and 9). We repair the individual to avoid performing

time-consuming WCET and energy analyses of solutions that violate the MO_S problem constraints. We perform jump correction, WCET analysis, and energy analysis after the SPM size constraint is satisfied (Lines 10 and 11).

After initializing and evaluating the initial population, we initialize the hyperparameters required to run the metaheuristic algorithms (Line 12). As discussed in Subsections 4.4.1 and 4.4.2, each metaheuristic algorithm has its own set of hyperparameters. Metaheuristics are iterative in nature, and to bring the algorithm to fruition, appropriate stopping criteria need to be predefined. In this thesis, we (unless specified otherwise) consider two stopping criteria (Line 13). The first criterion is the maximum number of generations, and the second stopping criterion is the maximum number of generations for which the population remains the same. For example, let the second stopping criterion be set to 10. If all the individuals within the population remain the same for 10 consecutive generations, then the optimization is stopped.

At every iteration, we update each individual in the current population with the help of a global or local operator or both (Line 15). The operators vary depending on the metaheuristic algorithm under consideration. SPEA uses mutation and recombination operators (cf. Subsection 4.4.1), and FPA uses global and local pollination operators (cf. Subsection 4.4.2). We check if the updated individuals satisfy the ISPM and DSPM size constraints, repair them if the constraints are not satisfied, perform jump correction on valid MO_S problem individuals, and evaluate the updated individual using WCET and energy analyzers (Lines 16-20).

Based on the WCET and energy objective values, the algorithm uses a selection operator (cf. Section 4.4) based on Pareto dominance (cf. Definition 4.1) to collect the top-scoring solutions and pass them to the next generation (Line 21). After fulfilling the stopping criteria, the algorithm outputs the Pareto optimal solution set out of the final population (Line 22). We collect the Pareto fronts P times from each initial population and generate the Final Pareto optimal solution set (PF_F) using P Pareto optimal solution sets (Line 23). As Pareto fronts obtained using metaheuristic algorithms are approximations, we assume that the Pareto front derived using P different approximate Pareto fronts is the best possible approximation that we can get in polynomial time during this metaheuristic algorithm run.

5.4 Evaluation

In this section, we solve the proposed multi-objective static SPM allocation optimization MO_S using two metaheuristic algorithms—SPEA and FPA. Moreover, we compare the obtained MO_S solutions with ILP-based single-objective static SPM allocation SO_S and the baseline evaluation BL, where the whole code is executed from the Flash memory. As mentioned before, MO_S considers static allocation of basic blocks and global data objects. However, the state-of-the-art SO_S optimization considers static allocation of basic blocks.

Subsection 5.4.1 describes the evaluation setup used to solve the optimization problem and the parameters used for the metaheuristic algorithms. Subsection 5.4.2 presents the Pareto optimal solutions found after solving MO_S, SO_S, and BL. Subsection 5.4.3 compares the quality of MO_S, SO_S, and BL solutions using quality indicators described in Subsection 4.3.3. Subsection 5.4.4 presents compilation time results for MO_S and SO_S.

5.4.1 Evaluation Setup

The evaluations are performed on an Intel XEON Gold 6146 server, with 48 cores clocked at 3.2 GHz with 1.48 TiB RAM. We implemented the MO_S optimization and the metaheuristic algorithms required to solve the optimization within the WCET-aware C Compiler (WCC) (cf. Chapter 3). The evaluations are conducted on the ARM Cortex-M0 microcontroller's fork of WCC. We assume that the benchmarks will fit within the Flash memory. The static WCET and energy analyses for MO_S are performed using a WCET analyzer called `aiT` v21.04i [Abs24] and an energy analyzer called `EnergyAnalyzer` v21.04i [Tea20].

We use the `MRTC` benchmark suite [GBEL10] with loop bound annotations from the `TACLeBench` project [FAH+16] for evaluation purposes. We compiled, analyzed, and evaluated 28 benchmarks from `MRTC` in this section. The code is generated by the WCC compiler using the standard `-O2` optimization flag. We adjust the ISPM and DSPM size individually to 60% relative to the benchmark size to increase the pressure on the optimization, even for small benchmarks.

We solve the MO_S problem using the FPA and SPEA algorithms for $P = 5$ different initial populations to reduce the influence of the initial population on the final Pareto front (cf. Algorithm 5.1). We set a timeout value of 20 h for each benchmark, and the compiler terminates the optimization if it is not complete. Within the time limit, we evaluate MO_S using both FPA and SPEA algorithms, and for each algorithm, the evaluation is repeated $P = 5$ times. If the optimization is not complete within the time limit, we derive a Pareto front from the current running generation and terminate the optimization.

As mentioned in the chapter's introduction and Section 5.1, SO_S is an ILP-based single-objective static SPM allocation optimization within WCC that minimizes WCET, so we perform energy analysis for its final solution to obtain the respective energy consumption value. For SO_S, we first create the ILP model by conducting two WCET analyses: one with the entire code stored in the Flash and another with it in SPM. The ILP problem is solved using an ILP solver called `Gurobi` [Gur24], which is integrated within WCC. After solving the ILP problem, we perform WCET and energy analyses on the final solution. Since SO_S only requires three WCET analyses and one energy analysis, we set a timeout value of 2 h for each benchmark to solve the SO_S problem.

For metaheuristic algorithm-based optimizations, the total compilation time includes the duration needed to run the metaheuristic algorithm, generate individuals, and sequentially perform WCET and energy analyses. As individ-

Table 5.1.: Values for the FPA algorithm's hyperparameters

Hyperparameters	Values
Switch probability (p_s)	0.8
Scaling factor (γ)	0.1
Positive integer for standard gamma function (λ)	1.5
Size of the population	10
Maximum number of generations	50
Maximum number of generations with same population	10

Table 5.2.: Values for the SPEA algorithm's hyperparameters

Hyperparameters	Values
Crossover probability (p_c)	0.8
Mutation probability (p_m)	0.2
Size of the population	10
Size of the external set	10
Maximum number of generations	50
Maximum number of generations with same population	10

uals within a given population of a metaheuristic algorithm are independent of each other, we could parallelly carry out their analysis on multiple threads on the server, and it could further reduce the required compilation times for metaheuristic algorithms. If we parallelize the analysis of each individual on the server, the compilation time required by each benchmark would depend on the number of available threads at any given time. Therefore, we choose to solve the optimization problem for each benchmark and carry out all corresponding analyses sequentially on a single thread on the server. For ILP-based optimizations, the compilation time encompasses the time required to build an ILP model, conduct necessary WCET and/or energy analyses, and solve the ILP problem. Additionally, the total compilation time also accounts for the time taken by WCC to generate code, perform code transformations, and complete all required steps involved in solving these optimizations.

While solving MO_S using FPA, we use the following hyperparameter values: As the probability of the local pollination is higher than global pollination in nature, the switch probability between them is set to $p_s = 0.8$. According to parametric studies in [RYDSP15, YKH14], the positive integer for the standard gamma function $\lambda = 1.5$ and the scaling factor $\gamma = 0.1$ work well for most cases. We considered the size of the population in each generation as 10. The first stopping criterion—the maximum number of generations—is set to 50, and the second stopping criterion—the maximum number of generations for which the population remains the same—is set to 10. Table 5.1 summarizes all the hyperparameter values for FPA used to solve MO_S in this section.

For solving MO_S using SPEA, we use the following hyperparameter values: The size of the population for each generation and the size of the external

population set is fixed to 10. As the probability of crossover is higher than mutation during evolution, we set the crossover probability p_c for SPEA as 0.8 and the mutation probability p_m as 0.2. We use single-point crossover and multi-bit mutation strategies while running SPEA (cf. Subsection 4.4.1). The stopping criteria for both FPA and SPEA algorithms have the same values. Table 5.2 summarizes all the hyperparameter values for the SPEA algorithm used to solve the MO_S problem in this section. The results obtained during these evaluations are valid for the algorithm parameters described above.

5.4.2 Pareto Fronts

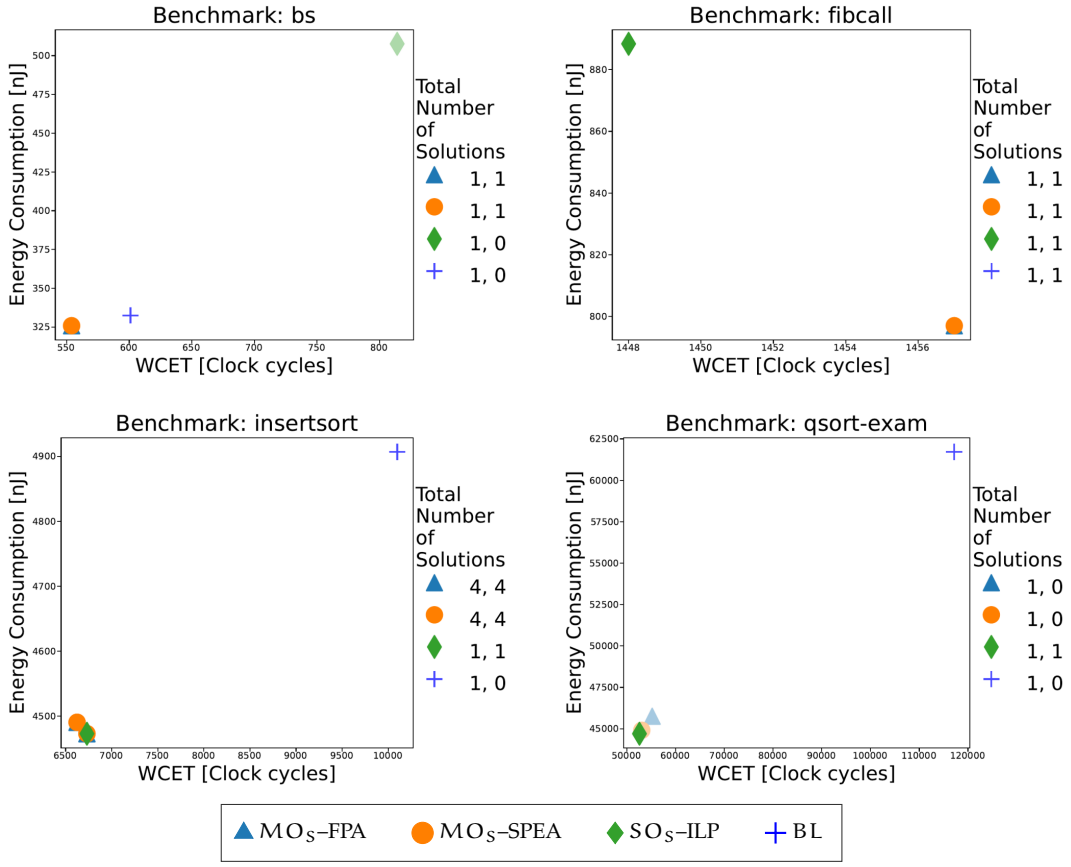
MO_S is a multi-objective optimization that aims to simultaneously minimize the program's WCET and energy consumption. In this section, we compared MO_S with an ILP-based single-objective optimization SO_S. As SO_S is a single-objective optimization, we will always have one solution on its respective Pareto front. Moreover, we compare MO_S with a baseline evaluation BL, which also has only one solution by definition.

As mentioned in Subsection 4.3.2, obtaining the true Pareto front for such multi-objective optimization problems is ambitious [KTZ06]. So, we realistically assume that the true Pareto front is unknown and construct an approximated True Pareto Front (PF_T) for each benchmark from the union of solutions obtained from MO_S–FPA, MO_S–SPEA, SO_S–ILP, and BL evaluation runs, i.e., $PF_T = \{p_i \mid \forall p_i \nexists c_j \in (PF_{F_{MO_S}} \cup PF_{F_{SO_S}} \cup PF_{F_{BL}}) \prec p_i\}$. p_i is the i^{th} non-dominated solution in PF_T; $PF_{F_{MO_S}} = PF_{F_{MO_S-FPA}} \cup PF_{F_{MO_S-SPEA}}$ is the union of final Pareto optimal solutions obtained during the MO_S optimization run using FPA ($PF_{F_{MO_S-FPA}}$) and SPEA ($PF_{F_{MO_S-FPA}}$); $PF_{F_{SO_S}}$ is the single solution obtained after solving the single-objective SO_S; $PF_{F_{BL}}$ is the single solution from the BL evaluation. For brevity's sake, hereafter, we refer to the Final Pareto Front obtained at the end of each optimization run only as PF_F.

Figure 5.3 presents the solutions found by MO_S–FPA, MO_S–SPEA, SO_S–ILP, and the BL evaluation. For brevity's sake, we present the solutions for only four benchmarks in this subsection. The figures for all the remaining benchmarks are in Appendix A. In Figure 5.3, we present 2D scatter plots for Pareto fronts obtained for four benchmarks—`bs`, `fibcall`, `insertsort`, and `qsort-exam`. The title of each subplot is the name of the respective benchmark. The legend at the bottom of the figure presents the optimization run to which a particular solution belongs. The legend at the right side of each subplot shows the total number of solutions on PF_F and PF_T for each optimization run. The PF_T for each benchmark is represented with the markers and their respective colors from the legend, and the fainter version of those colors represents the respective Pareto front PF_F returned by each approach.

For `bs`, MO_S–FPA and MO_S–SPEA contributed one solution each to their respective PF_Fs, which formed the final PF_T. However, it is important to note that SO_S provided a worse solution than the BL solution. Similarly, for

Figure 5.3.: Pareto fronts for four benchmarks using MO_S, SO_S, and BL



`janne_complex`, the SO_S solution had a slightly worse WCET than the BL solution (cf. Figure A.2). These are the only two cases where SO_S provided a worse solution than the BL evaluation. This scenario can occur when the incurred jump correction costs outweigh the benefits of static allocation. However, it is important to note that these two benchmarks are extremely small, and the additional jump correction code was added to generate the SO_S solution. For `bs`, SO_S and BL solution's code sizes are 130 and 94 bytes; for `janne_complex`, SO_S and BL solution's code sizes are 90 and 62, respectively. Moreover, the increase in WCET due to SO_S is slight as well. For `bs`, SO_S and BL solution's WCET values are 814 and 601 cycles; for `janne_complex`, SO_S and BL solution's WCET values are 1659 and 1619, respectively. Therefore, it could imply that the SO_S's ILP model could not accurately account for jump correction costs for `bs` and `janne_complex`. However, as SO_S uses only two evaluations to build the ILP model and tries to estimate the jump correction costs, some estimation errors are expected in such scenarios. Using such an ILP model, we cannot precisely predict the exact WCET value of an individual without performing static WCET analysis.

In Figure A.3, for `lcdnum` and `lms`, MO_S–FPA yielded solutions with higher objective values than the BL. Consequently, we can deduce that these MO_S–

Table 5.3.: Comparison between the average number of solutions (in %) on the final PF_T from the respective PF_F of each optimization approach

MO_S		SO_S	BL
FPA	SPEA	ILP	
36.36	60.34	39.29	7.14

FPA evaluation runs never encountered an individual with all basic blocks in the Flash memory (the BL condition). If the MO_S–FPA evaluation run had encountered such a BL solution, MO_S–FPA would have chosen the BL solution over the solutions worse than BL shown in Figure A.3. Furthermore, neither SO_S nor MO_S incorporates any constraints on objective values, where the obtained solution's objective values are constrained by the BL's WCET and energy values. If such constraints were applied to SO_S and MO_S, we could eliminate such situations.

In the case of `fibcall`, all MO_S, SO_S, and BL solutions are on the final PF_T. The MO_S and the BL solutions are the same, i.e., the MO_S could not find any better solution than the original code. However, as SO_S only optimizes for WCET, SO_S was able to find another solution with a lower WCET value and higher energy consumption compared to MO_S and BL solutions. For `insertsort`, all solutions from MO_S–FPA, MO_S–SPEA, and SO_S–ILP are on the final PF_T; all the obtained solutions from the optimizations are better than the BL solution. In the case of the `qsort-exam` benchmark, only SO_S–ILP contributed to the final PF_T; both MO_S–FPA and MO_S–SPEA were not able to find better solutions than SO_S.

From overall evaluations and obtained Pareto fronts (cf. Figure 5.3 and Appendix A), we can see that: MO_S–FPA contributed solutions to the final PF_T for 13 benchmarks; MO_S–SPEA contributed solutions to PF_T for 15 benchmarks; SO_S–ILP contributed solutions to PF_T for 11 benchmarks; BL contributed solutions to PF_T for 2 benchmarks. Therefore, we can say that MO_S performed slightly better than single-objective SO_S optimizations. Moreover, contrary to SO_S's single solutions, MO_S is able to provide more than one solution on the final Pareto front for most benchmarks.

Furthermore, we calculated the total number of solutions provided by each optimization on PF_T and averaged it over all the benchmarks. Table 5.3 provides the percentage of the number of solutions on PF_T, on average, out of the solutions on PF_F for each optimization. SO_S, on average, provided 39.29% of its PF_F solutions on the final PF_T. The BL evaluation, on average, provided 7.14% of its PF_F solutions on the final PF_T. However, these numbers are skewed towards single-solution-producing approaches like SO_S and BL. For example, in the case of the BL evaluation, we got solutions on PF_T for 2 out of 28 benchmarks, accounting for 7.14% of BL solutions.

As mentioned before, MO_S–FPA and SO_S contributed solutions to the final PF_T for 13 and 11 benchmarks, respectively. Consequently, MO_S–FPA performed slightly better than SO_S, but only 36.36% of its solutions ended on

Table 5.4.: Quality Indicators for MO_S, SO_S, and BL optimization approaches

	$\mathcal{C}$				NR				RNS				HV			
	MO_S		SO_S	BL	MO_S		SO_S	BL	MO_S		SO_S	BL	MO_S		SO_S	BL
Benchmarks	FPA	SPEA	ILP		FPA	SPEA	ILP		FPA	SPEA	ILP		FPA	SPEA	ILP	
adpcm	**0**	1	1	1	**1**	0	0	0	**1**	0	0	0	**0.999**	0.977	0	0
bs	**0**	**0**	1	1	**0.5**	**0.5**	0	0	**1**	**1**	0	0	**0.554**	**0.554**	0.002	0.514
bsort100	**0**	1	1	1	**1**	0	0	0	**1**	0	0	0	**0.998**	0.981	0.992	0
cnt	**0**	1	1	1	**1**	0	0	0	**1**	0	0	0	**0.939**	0.56	0.489	0
compress	1	1	**0**	1	0	0	**1**	0	0	0	**1**	0	0.444	0.523	**0.991**	0
crc	1	**0**	1	1	0	**1**	0	0	0	**1**	0	0	0.664	**0.996**	0	0
edn	1	**0**	**0**	1	0	**0.75**	0.25	0	0	**1**	**1**	0	0.973	**0.993**	0.973	0
expint	1	**0**	1	1	0	**1**	0	0	0	**1**	0	0	0.873	**0.887**	0	0.046
fac	**0**	**0**	1	1	**0.5**	**0.5**	0	0	**1**	**1**	0	0	**0.639**	**0.639**	0.001	0.219
fdct	**0**	**0**	1	1	**0.5**	**0.5**	0	0	**1**	**1**	0	0	**0.859**	**0.859**	0.512	0
fibcall	**0**	**0**	**0**	**0**	**0.25**	**0.25**	**0.25**	**0.25**	**1**	**1**	**1**	**1**	**0.492**	**0.492**	0.005	**0.492**
fir	**0**	1	1	1	**1**	0	0	0	**1**	0	0	0	**0.997**	0.892	0.891	0
insertsort	**0**	**0**	**0**	1	**0.444**	**0.444**	0.111	0	**1**	**1**	**1**	0	**0.815**	**0.815**	0.796	0
janne_complex	**0**	**0**	1	1	**0.5**	**0.5**	0	0	**1**	**1**	0	0	**0.52**	**0.52**	0.003	0.405
jfdctint	0.5	**0**	1	1	0.333	**0.667**	0	0	0.5	**1**	0	0	0.897	**0.903**	0.145	0
lcdnum	1	**0**	1	**0**	0	**0.5**	0	**0.5**	0	**1**	0	**1**	0.341	**0.467**	0.002	0.446
lms	1	1	**0**	1	0	0	**1**	0	0	0	**1**	0	0.013	0.049	**0.999**	0.006
matmult	1	1	**0**	1	0	0	**1**	0	0	0	**1**	0	0.963	0.981	**0.998**	0
minver	**0.333**	0.5	1	1	**0.667**	0.333	0	0	**0.667**	0.5	0	0	**0.951**	0.949	0	0
ns	**0**	**0**	1	1	**0.5**	**0.5**	0	0	**1**	**1**	0	0	**0.959**	**0.959**	0.634	0
prime	1	0.333	**0**	1	0	**0.667**	0.333	0	0	0.667	**1**	0	0.616	**0.649**	0.571	0.012
qsort-exam	1	1	**0**	1	0	0	**1**	0	0	0	**1**	0	0.887	0.966	**0.985**	0
qurt	1	1	**0**	1	0	0	**1**	0	0	0	**1**	0	0.18	0.239	**0.944**	0.037
recursion	0.333	**0.25**	1	1	0.4	**0.6**	0	0	0.667	**0.75**	0	0	0.339	**0.606**	0.15	0.15
select	**0**	1	1	1	**1**	0	0	0	**1**	0	0	0	**0.91**	0.568	0.837	0
sqrt	0.75	**0**	1	1	0.125	**0.875**	0	0	0.25	**1**	0	0	0.501	**0.587**	0.111	0.111
st	1	1	**0**	1	0	0	**1**	0	0	0	**1**	0	0.141	0.232	**0.999**	0
ud	1	1	**0**	1	0	0	**1**	0	0	0	**1**	0	0.124	0.256	**0.881**	0.008

PF_T. On the contrary, MO_S–SPEA provided the most—60.34%—number of its solutions to PF_T. Therefore, we can say that MO_S performed slightly better than SO_S, and MO_S–SPEA performed better than MO_S–FPA.

5.4.3 Quality Indicators

To better analyze the quality of obtained solutions, we consider four quality indicators—Coverage ($\mathcal{C}$), Non-dominance Ratio (NR), Ratio of Non-dominated Solutions (RNS), and Hypervolume (HV)—explained in detail in Subsection 4.3.3.

Table 5.4 presents quality indicator values for all benchmarks for MO_S, SO_S, and BL approaches. For each benchmark, the approach with quality indicators in bold performed better (or nondominant) than other approaches.

For example, in the case of adpcm, MO_S–FPA performed better than other approaches in terms of all quality indicators. For the bs benchmark, MO_S–FPA and MO_S–SPEA have the same quality indicator values. Therefore, the quality of solutions obtained by MO_S–FPA and MO_S–SPEA is indifferent to each other, but it is better than SO_S and BL.

We condense this information from Table 5.4 into Table 5.5 by counting the total number of benchmarks with the best (or nondominated) quality indicators. Table 5.5 summarizes the total number of benchmarks with best or nondominated $\mathcal{C}$, NR, RNS, and HV. In this table, the numbers shown outside the bracket are the total number of benchmarks with the best quality indicators,

Table 5.5.: Total number of benchmarks with best (or nondominated) quality indicators for each optimization approach

Quality Indicators	MO_S FPA	MO_S SPEA	SO_S ILP	BL
$\mathcal{C}$	6 (7)	5 (9)	8 (3)	0 (2)
NR	6 (7)	7 (8)	7 (2)	0 (2)
RNS	6 (7)	5 (9)	8 (3)	0 (2)
HV	6 (7)	8 (7)	7 (0)	0 (1)

and the numbers inside the bracket are the number of benchmarks with nondominated Pareto optimal solutions compared to other approaches.

For example, in Table 5.4, `fac` has the same quality indicator values for MO_S–FPA and MO_S–SPEA; the benchmarks with such quality indicators are quantified in Table 5.5 by the value inside the bracket. For `insertsort`, we have indifferent $\mathcal{C}$ and RNS values for MO_S–FPA, MO_S–SPEA, and SO_S–ILP. However, MO_S–FPA and MO_S–SPEA provide four solutions each on the final Pareto front PF_T compared to the one solution provided by SO_S–ILP (cf. Figure 5.3). Therefore, the NR and HV values for SO_S–ILP were worse than MO_S–FPA and MO_S–SPEA. Consequently, `insertsort` has indifferent NR and HV only for MO_S–FPA and MO_S–SPEA approaches.

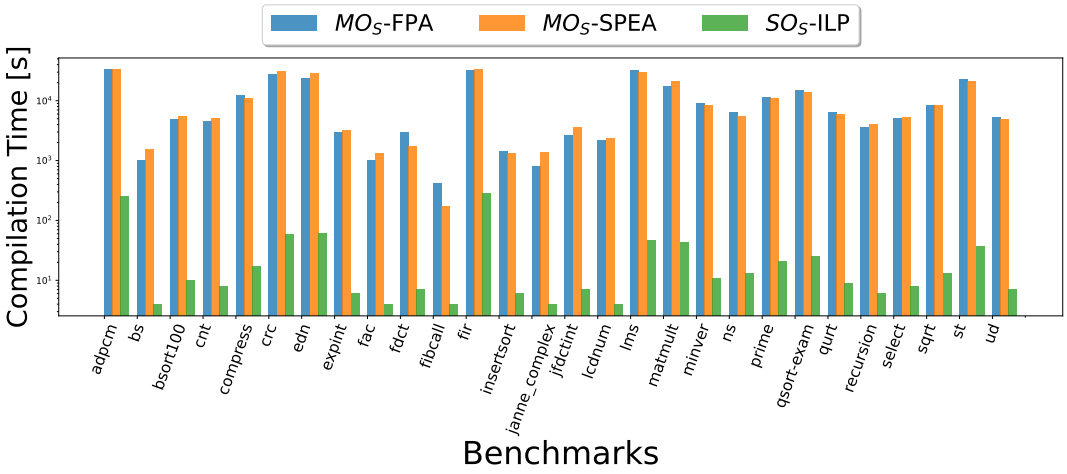
In Table 5.4, we can see for the `matmult` benchmark that SO_S provided the only solution on the final PF_T; the benchmarks with such best quality indicators are quantified in Table 5.5 by the value outside the bracket. Tables 5.4 and 5.5 show that, on average, MO_S–FPA and MO_S–SPEA provided better or indifferent quality solutions for more benchmarks compared to SO_S.

SO_S–ILP provided better or indifferent $\mathcal{C}$ and RNS indicators for a total of 11 benchmarks. For 7 benchmarks, SO_S provided better solutions in terms of NR and HV, and for 2 benchmarks, SO_S had indifferent NR quality indicators. On the other hand, MO_S–FPA provided better or indifferent quality indicators for a total of 13 benchmarks, whereas MO_S–SPEA provided better or indifferent $\mathcal{C}$ and RNS indicators for 14 benchmarks and better or indifferent NR and HV metrics for a total of 15 benchmarks.

For `fibcall` and `lcdnum`, the BL solutions were also on the final PF_T (cf. Figures 5.3 and A.3). The `fibcall` case is explained in detail in Subsection 5.4.2. In the case of `lcdnum`, MO_S–SPEA and BL contributed one solution each on the final PF_T. MO_S–FPA and SO_S–ILP did not contribute any solutions on PF_T. Similar to the `bs` case (cf. Figure 5.3), for `lcdnum`, MO_S–FPA provided a solution worse than BL (cf. Subsection 5.4.2).

On average, MO_S–FPA and MO_S–SPEA provided better or indifferent quality solutions for more benchmarks than SO_S, and both MO_S and SO_S outperformed BL. Moreover, for MO_S, SPEA slightly outperformed FPA by providing more solutions on PF_T (cf. Tables 5.3 and 5.5).

Figure 5.4.: Compilation times for MO_S and SO_S optimization runs



5.4.4 Compilation Times

In this subsection, we compare the compilation time required to solve MO_S and SO_S with each other. Figure 5.4 shows the comparison of the compilation time required by MO_S–FPA, MO_S–SPEA, and SO_S–ILP optimization runs in the form of a bar plot. The x-axis represents all the evaluated benchmarks, and the y-axis shows the compilation time required to finish the optimization. For the BL evaluation, we are just conducting WCET and energy analysis of the code compiled by WCC. Therefore, for brevity's sake, we do not consider BL in Figure 5.4 as it is not an optimization.

From the figure, we can see that MO_S was more time-consuming than SO_S during this evaluation. For MO_S, FPA was slightly faster compared to SPEA. However, SPEA provided better quality solutions, as discussed in previous subsections. SO_S–ILP took, on average, 0.59 min to finish the optimization, whereas MO_S–FPA and MO_S–SPEA took, on average, 2.95 h and 3 h to finish the optimization runs, respectively.

As mentioned in Section 5.4.1, SO_S requires two WCET analyses to build its ILP model; the first WCET analysis is done by keeping all basic blocks in the Flash to collect respective Flash-related ILP constants; the second WCET analysis is done by temporarily placing all basic blocks in SPM to collect respective SPM-related ILP constants. Next, the ILP model is used to solve the SO_S optimization problem using an ILP solver called `Gurobi` [Gur24], which is integrated within WCC. Finally, the WCET and energy analyses are performed on the final solution obtained after solving the SO_S problem. Therefore, the compilation time for SO_S encompasses the time required to perform four evaluations and to solve the ILP problem.

On the other hand, the MO_S problem, while using FPA or SPEA, requires a WCET and energy analysis for each individual (2 evaluations). As mentioned

in Subsection 5.4.1, the size of the population is 10, the maximum number of generations is 50, and we perform evaluations for $P = 5$ different initial populations. Therefore, we need to perform a maximum of $2 * 10 * 50 * 5 = 5000$ evaluations to finish the optimization run using each metaheuristic algorithm. If the second stopping criterion—the maximum number of generations with the same population—is reached before meeting the first criterion, the optimization might require fewer than 5000 evaluations for each metaheuristic algorithm.

WCET and energy analyses can be time-consuming based on the underlying benchmark size. Consequently, it was expected that MO_S would be time-consuming compared to SO_S. The compilation time required by SO_S can be more than MO_S only if the ILP solver takes a very long time to solve the ILP. However, we did not encounter such a scenario in these evaluations.

5.5 Conclusion

In this chapter, we proposed a compiler-level multi-objective static SPM allocation optimization (MO_S) that aims to minimize the WCET and energy consumption of the program simultaneously. We used two metaheuristic algorithms—FPA and SPEA—to solve the MO_S optimization problem. The architecture-specific parts of the optimization, code generation, and analyses during evaluations were done for ARM Cortex-M0.

During evaluations, we compare MO_S with a single-objective ILP-based static SPM allocation optimization (SO_S) and a baseline evaluation BL to evaluate MO_S's solutions and performance. Evaluations show that the quality and quantity of solutions obtained using MO_S are slightly better than SO_S, and MO_S clearly outperformed BL. Unlike SO_S, MO_S has provided more than one Pareto optimal solution for many benchmarks. Moreover, for MO_S, the SPEA algorithm performed better than FPA during these particular evaluations. However, due to the random nature of the algorithms, we cannot definitely say if SPEA will always perform better than FPA for the MO_S problem. Furthermore, we saw that MO_S is a time-consuming optimization that requires multiple evaluations to bring the optimization to fruition. Therefore, the time taken by each evaluation serves as a huge bottleneck while solving these multi-objective optimizations, and the next two chapters deal with this bottleneck.

In the next chapter, we propose an approximation model that aims to estimate the WCET and energy consumption of an individual while running the metaheuristic algorithms. To deal with the compilation time bottleneck, we use the proposed approximation model instead of the analyzer to determine an individual's objective value. In Chapter 7, we propose two more approaches to reduce the search space dimension that might lead to a reduction in the required number of evaluations and, eventually, the required compilation time.

6

Approximating WCET and Energy for Fast Multi-Objective Static SPM Allocation

In Chapter 5, we exploited metaheuristic algorithms to solve compiler-level multi-objective static SPM allocation problems. However, the compilation time required to solve the optimization problem was high due to the iterative nature of the metaheuristic algorithms. The following parameters can serve as a bottleneck while solving such a multi-objective optimization problem:

- The size of the search space: The solution space of our optimization problem depends on the code size. Therefore, as the code size increases, the dimension of the search space can increase.
- The lack of a general equation describing objective functions: We do not know the mathematical equation that represents the WCET and the energy consumption of a general program.
- The time required for analyzing objective functions: Due to a lack of a general equation describing WCET and energy consumption, we need to perform WCET and energy analyses to estimate objective values, which can be time-consuming.
- In real-life scenarios, we cannot let the optimization run wild without any time limit. Therefore, we cannot evaluate every possible solution for such an optimization problem.

In Chapter 5, we used a static analysis tool called `aiT` [Abs24] to determine the WCET of a program at compile time. Similarly, the energy consumption of a program was estimated using an energy analysis tool called `EnergyAnalyzer` [Tea20], which is integrated tightly within WCC. Performing WCET and energy analyses using these analyzers can be time-consuming, depending on the program size. While performing metaheuristics-based multi-objective optimization, analyzing all the individuals in the population using such analyzers can become time-exhaustive. Therefore, in this chapter, we propose a novel approximation model to approximate objective values and use it instead of the time-exhaustive analyzers to analyze the population.

The approximation model proposed in this chapter is built based on the data collected from four initial WCET and energy analyses of the program with different code allocations. We conduct the four initial analyses using the WCET and energy analyzers that are integrated within WCC. The approximation model approximates WCET and energy objectives to bring the metaheuristic algorithm to fruition. Finally, the compiler performs safe analyses on the final solution set using the static analyzers to present correct WCET and energy consumption values.

The approximation model and the approximation-aware multi-objective static SPM allocation described in this chapter were presented at the International Conference on Real-Time Networks and Systems (RTNS) in 2022 [JF22].

Similar to the $\mathrm{MO_S}$ optimization problem from Chapter 5, in this chapter, we solve the multi-objective static SPM allocation problem using two metaheuristic algorithms—FPA and SPEA. For the sake of brevity, we refer to the approximation-aware multi-objective static SPM allocation optimization proposed in this chapter as $\mathrm{MO_{Approx}}$. The $\mathrm{MO_{Approx}}$ approach uses the approximation models proposed in the upcoming sections to approximate the WCET and the energy consumption of the solution vector instead of the respective analyzers.

The approximation models proposed in this chapter can over- or underestimate the WCET and energy consumption objectives while solving $\mathrm{MO_{Approx}}$ optimization. However, these approximated objectives are only used to guide the metaheuristic algorithms towards fruition. We analyze the final Pareto optimal solutions provided by metaheuristic algorithms using `aiT` and `EnergyAnalyzer`. As we do not use the approximation models to obtain the WCET and energy consumption values for the final solutions, they do not harm the sanctity of the real-time nature of WCC's output. The key contributions of this chapter are:

- We proposed an approximation model to approximate the program's WCET and energy consumption.
- We integrated the proposed approximation model into multi-objective static SPM allocation optimization ($\mathrm{MO_{Approx}}$).
- We solve $\mathrm{MO_{Approx}}$ optimization using FPA and SPEA algorithms.

During evaluations, we compare the results from $\mathrm{MO_{Approx}}$, $\mathrm{MO_S}$ (multi-objective static SPM allocation proposed in Chapter 5), $\mathrm{SO_S}$ (single-objective ILP-based static SPM allocation from Chapter 5), and BL (the baseline evaluation from Chapter 5). We compare $\mathrm{MO_{Approx}}$, $\mathrm{MO_S}$, and $\mathrm{SO_S}$ in terms of compilation times; $\mathrm{MO_{Approx}}$, $\mathrm{MO_S}$, $\mathrm{SO_S}$, and BL are compared in terms of the quality of the obtained final solution set; $\mathrm{MO_{Approx}}$ and $\mathrm{MO_S}$ are compared to determine approximation errors due to the use of the proposed model. The key results from the evaluations are:

- Using MO_{Approx}, we achieved, on average, a 94.89% decrease in compilation time compared to MO_S.
- The approximation errors during the MO_{Approx} run were in the acceptable range of 4% - 7%, on average.
- The quality of the solutions provided by MO_{Approx} is better than the ones from MO_S.
- MO_{Approx}–SPEA performed better than all the other approaches during the evaluations conducted in this chapter.

This chapter is organized as follows: Section 6.1 provides an overview of the related background work. Section 6.2 formulates the SPM allocation-based multi-objective optimization problem solved in this chapter. Section 6.3 presents models for the approximation of the objective functions. Section 6.4 briefly discusses the metaheuristic algorithms to solve MO_S and MO_{Approx}. Section 6.5 presents the evaluation results, and Section 6.6 concludes the chapter.

6.1 Related Work

Some research has been done to exploit machine learning-based approaches to predict the WCET and energy consumption of the code. Huybrechts et al. [HMH18] proposed a machine learning-based hybrid approach using different regression models to predict the program's WCET. Bonenfant et al. [BCDMS17] predicted WCET by combining static source code analysis and machine learning techniques at the source code level.

Altenbernd et al. [AGLS16] proposed a method to identify a source code-level linear timing model that predicts WCET for a certain hardware configuration and compiler. Muts et al. [MF21b] presented a machine learning-based method to build a surrogate model to predict a program's WCET at compile-time instead of using time-consuming WCET analysis. They used their prediction model to solve the compiler-level multi-criterial function inlining problem. Their prediction model is independent of the source code and assembly code features. However, they observed that for benchmarks with larger objective space, their classifier either resulted in poor predictions or required many training points, i.e., it required many WCET analyses to build the model.

Using power profile results, hardware performance characteristics, and machine learning models, Sachan et al. [SG21] determined the optimal settings for compiler optimization to minimize energy consumption. Mishra et al. [MZLH15] proposed a probabilistic graphical machine learning-based system that can provide online estimates of an application's performance and its online power consumption. Muts et al. [MF21a] presented a search space reduction method based on the selection of optimization-related important

features. They combined it with a machine learning-based prediction model to quickly predict the program's WCET and energy consumption while performing multi-objective function inlining. However, they were also required to perform multiple time-consuming analyses at the beginning to gather WCET and energy consumption-related data to have the training set to build the prediction model.

The accuracy of these approaches depends on the data validity and the data size used to build the machine learning model. Also, the model's accuracy depends on the hyperparameters used for the respective machine learning models and their proper tuning. In contrast, our approach uses a fixed number of evaluations, i.e., four, to build the approximation model and does not depend on any hyperparameters.

6.2 Problem Formulation

Similar to Chapter 5, we formulate a compiler-level multi-objective optimization problem that performs SPM allocation to minimize the program's WCET and energy consumption (MO_{Approx}) in this section. However, the objectives of the MO_{Approx} problem are approximated using the approximation model proposed in Section 6.3. The problem formulation for MO_{Approx} looks similar to the MO_S problem, which is discussed in detail in Section 5.2.

As metaheuristic algorithms are iterative population-based algorithms, evaluating all the individuals at every iteration is necessary to drive the optimization to fruition. These individuals in the population represent the decision variables from the search space. However, WCET and energy analyses are time-consuming processes. Therefore, given the large number of evaluations needed, finding approximated Pareto optimal solutions to MO_S, even for small programs, could take a long time (cf. Section 5.4).

To tackle this issue in this chapter, we instead formulate an approximated multi-objective optimization problem (MO_{Approx}) as follows.

$$\begin{aligned} \min_{x} \quad & \widetilde{F}(x) = (\widetilde{F_1}(x), \widetilde{F_2}(x)) \\ \text{subject to} \quad & Q_1(x_{1:\tilde{d}}) \leqslant 0 \text{ and } Q_2(x_{(\tilde{d}+1):d}) \leqslant 0, \end{aligned} \tag{6.1}$$

where $\widetilde{F_1}(x)$ and $\widetilde{F_2}(x)$ represent the approximated WCET and the approximated energy consumption minimization objectives, respectively. The approximated objective function $\widetilde{F}(x)$ maps the search space $X \subset \{0,1\}^d$ to the objective space $Y \subset \mathbb{R}^2$.

As we are performing multi-objective static SPM allocation, the rest of the problem formulation is identical to Section 5.2 by definition. The search space X consists of $x \in X$—a d-dimensional binary decision variable vector. The subvector $x_{1:\tilde{d}}$ decides whether basic blocks (b_i) will be placed in ISPM or Flash, i.e.,

$$\forall i = \overline{1, \tilde{d}} \quad x_i = \begin{cases} 1, & \text{if } b_i \text{ is placed in ISPM,} \\ 0, & \text{otherwise} \end{cases} \tag{6.2}$$

The subvector $x_{(\tilde{d}+1):d}$ decides whether global data variables ($\mathcal{G}_g$) will be placed in DSPM or Flash, i.e.,

$$\forall g = \overline{(\tilde{d}+1), d} \quad x_g = \begin{cases} 1, & \text{if } \mathcal{G}_g \text{ is placed in DSPM,} \\ 0, & \text{otherwise} \end{cases} \tag{6.3}$$

$\tilde{d}$ represents the total number of basic blocks in the program, and $(d - \tilde{d})$ is the total number of global data variables. The constraint condition $Q_1(x_{1:\tilde{d}}) \leqslant 0$ for the MO_{Approx} problem is the size of the ISPM, i.e.,

$$Q_1(x_{1:\tilde{d}}) = \sum_{i=1}^{\tilde{d}} B_i x_i + \Psi_J - S_{ISPM} \tag{6.4}$$

where B_i is the code size of basic block b_i, S_{ISPM} is the total size of the ISPM, and Ψ_J represents the extra cost incurred due to jump correction in terms of code size. Jump correction is discussed in detail in Subsection 3.1.3, and the jump correction cost Ψ_J is discussed in detail in Section 5.2.

The second constraint condition $Q_2(x_{(\tilde{d}+1):d}) \leqslant 0$ in Equation 6.1 represents the number of global data variables that can fit in the DSPM statically.

$$Q_2(x_{(\tilde{d}+1):d}) = \sum_{g=(\tilde{d}+1)}^{d} S_{\mathcal{G}_g} x_g - S_{DSPM} \tag{6.5}$$

where $S_{\mathcal{G}_g}$ is the size of the global data variable $\mathcal{G}_g$, and S_{DSPM} is the total size of the DSPM.

To solve MO_{Approx}, instead of performing WCET and energy analyses using respective analyzers, we evaluate the individuals by approximating the objective values. We present the approximation model for said objectives in the following section.

6.3 Approximation Model

In this section, we propose an approximation model necessary to approximate the objectives while performing multi-objective static SPM allocation. This section is further divided into the following subsections: Subsection 6.3.1 presents the individuals required to build the approximation model. Subsections 6.3.2 and 6.3.3 propose the approximation models for WCET and energy consumption objectives. Subsection 6.3.4 discusses the limitations of the proposed approximation model.

6.3.1 Individuals for the Approximation Model

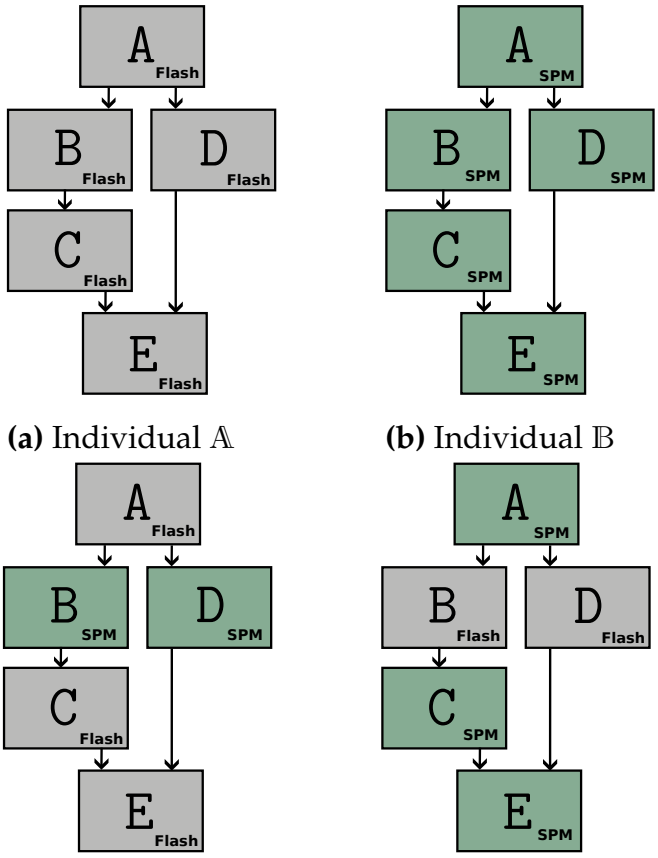
To build the approximation model, we first generate four distinct individuals referring to four specific code allocations, which are explained in detail later. The analysis of these four individuals using WCET and energy analyzers provides us with all the data necessary to build the approximation models for WCET and energy consumption of the program. We use the approximation model proposed in this section to analyze individuals iteratively from populations only while running metaheuristic algorithms. Once a metaheuristic algorithm provides the final Pareto optimal solution set, we analyze Pareto optimal individuals using WCET and energy analyzers to avoid minor approximation errors in the final output. This step ensures that WCC finally provides appropriately analyzed Pareto optimal solutions.

Let $\mathbb{A}, \mathbb{B}, \mathbb{C}, \mathbb{D} \in X$ represent four individuals necessary to build the approximation model, and $X \subset \{0,1\}^d$ is a set of all individuals.

- Individual $\mathbb{A}$ represents the case where we allocate all the basic blocks to the Flash memory.
- Individual $\mathbb{B}$ represents the case where we place all the basic blocks in SPM. As SPMs are small-sized memories, the size of SPM is hypothetically increased to place all basic blocks in SPM for individual $\mathbb{B}$.
- For individual $\mathbb{C}$, we allocate the first basic block to the Flash, the second to the SPM, and so on, i.e., each basic block is placed in the Flash and SPM in an alternating fashion, starting with the Flash memory.
- For individual $\mathbb{D}$, we allocate the first basic block to the SPM, the second to the Flash memory, and so on, i.e., each basic block is placed in the SPM and Flash in an alternating fashion, starting with the SPM.

We need basic block-level WCET and energy information to approximate the objective values for different configurations of individuals generated while running metaheuristic algorithms. Using Individuals $\mathbb{A}$ and $\mathbb{B}$, we can get the WCET and energy values when basic blocks are placed in Flash and SPM, respectively. However, the WCET and energy information provided by Individuals $\mathbb{A}$ and $\mathbb{B}$ represent the basic blocks unmodified by the jump correction code. If a basic block is allocated to SPM and its succeeding or preceding basic block is still in the Flash, we must add the jump correction code to fix the CFG (cf. Section 3.1.3). The modifications in basic blocks due to jump corrections are not captured by Individuals $\mathbb{A}$ and $\mathbb{B}$. Therefore, we use the WCET and energy information provided by Individuals $\mathbb{C}$ and $\mathbb{D}$ to approximate the objective values more precisely. We could have considered more variations of individuals to build a more precise approximation model. However, we want a faster way to approximate WCET and energy consumption,

Figure 6.1.: An example with four possible allocation cases covered by the individuals. We consider only five basic blocks, `A`, `B`, `C`, `D`, and `E`, in this example, where `A` has two successors (`B` and `C`). Let the decisions for the five basic blocks be arranged alphabetically as $(\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E})$ in the decision vector x.



(a) Individual $\mathbb{A}$ **(b)** Individual $\mathbb{B}$

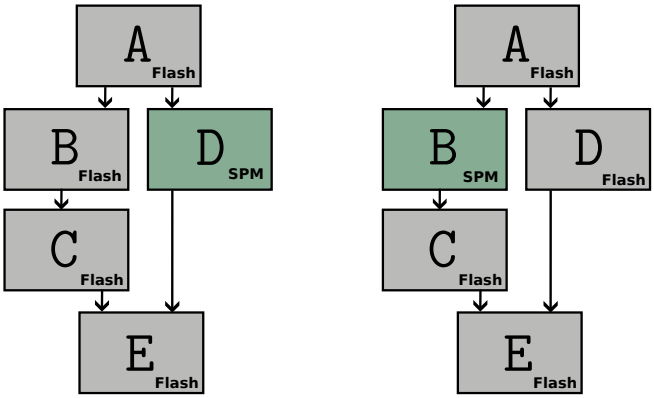
(c) Individual $\mathbb{C}$ **(d)** Individual $\mathbb{D}$

and having to do numerous evaluations upfront to build an approximation model would defeat that purpose.

Figure 6.1 provides an example with five basic blocks where their possible allocations covered by the four individuals $\mathbb{A}, \mathbb{B}, \mathbb{C}$, and $\mathbb{D}$ are shown. In this figure, we consider that the decisions for the five basic blocks in the decision vector x are arranged alphabetically as follows: $(\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E})$. The approximation model is able to consider the following cases using the analysis results of these four individuals.

- Individual $\mathbb{A}$ is used to approximate the objective values of basic blocks placed in the Flash.
- Individual $\mathbb{B}$ is used to approximate the objective values of basic blocks placed in the SPM.

Figure 6.2.: Two exemplary allocations for the CFG from Figure 6.1 that are not covered by the four individuals $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$, and $\mathbb{D}$



- The alternating placement of basic blocks within individuals $\mathbb{C}$ and $\mathbb{D}$ approximates the effect of jump correction on the basic block's objective value, i.e., the analysis of individuals $\mathbb{C}$ and $\mathbb{D}$ provides the extra cost in terms of WCET and energy consumption required to jump between the Flash and SPM.

Similar to individual $\mathbb{B}$, for individuals $\mathbb{C}$ and $\mathbb{D}$, we hypothetically increase the SPM size to accommodate all the basic blocks allocated to the SPM if they do not fit within the SPM. These four individuals are analyzed using WCET and energy analyzers, and the objective values of respective individuals are annotated within WCC at the basic block-level granularity.

All possible allocation scenarios for basic blocks with one successor are covered by these individuals. However, these four individuals do not cover all possible cases that we can encounter while performing static SPM allocation. The following cases are not covered by the four individuals:

- Not all the cases of basic block allocation with two successors might be represented by one of the four individuals $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$, and $\mathbb{D}$ from Figure 6.1. Figure 6.2 shows two cases not covered by these four individuals. Figure 6.2 shows the two cases where basic blocks `B` and `D` are in different memories. However, according to individuals $\mathbb{C}$ and $\mathbb{D}$, basic blocks `B` and `D` are always in the same memory. As the vector of basic blocks in Figures 6.1 and 6.2 are arranged alphabetically, i.e., (`A`, `B`, `C`, `D`, `E`), the basic blocks `B` and `D` are always in the same memory. Therefore, the two cases from Figure 6.2 cannot be covered by individuals $\mathbb{C}$ and $\mathbb{D}$.

- Not all the cases of basic block allocation with a function call might be represented by one of the four individuals $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$, and $\mathbb{D}$ from Figure 6.1. Some possible allocations between the basic block calling the function and the first basic block of that function might be covered by the four analyzed individuals. However, we cannot guarantee that all possible cases can be covered by them.

Therefore, while building the approximation model, we need to add extra costs within the model such that all such scenarios not covered by the four analyzed individuals are also considered.

6.3.2 Approximation Model for WCET

In this subsection, we propose an approximation model for the WCET objective and provide its detailed mathematical formulation. As mentioned in Section 6.2, the individual decision vector $x \in X$ consists of both basic blocks and global data objects. Therefore, individuals $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$, and $\mathbb{D}$ convey allocation information for basic blocks and global data objects while building the approximation model. The first six terms in the approximation model focus on basic blocks, whereas the last two focus on the global data objects.

Let $W_{(f,i)}$ represent the WCET value of basic block b_i $\forall i = \overline{1, \tilde{d}}$ of the individual $f \in X$, and $\tilde{d}$ is the total number of basic blocks within a program. Let $\mathbb{E}$ represent the set of edges of the control flow graph of the program, where $e_{(i,k)} \equiv (b_i, b_k) \in \mathbb{E}$ indicates that the basic block b_k is reached from the basic block b_i. Next, we define all the terms required to build the approximation model that approximates the program's WCET:

1. $f_1(x)$ is the sum of the WCETs of all basic blocks that are placed in the Flash according to the decision vector x, where $W_{(\mathbb{A},i)}$ represents the WCETs of the i^{th} basic block of the individual $\mathbb{A}$. As mentioned before, $x_i \in \{0, 1\}$ represents a decision variable for the basic block b_i, and $\overline{x_i} = \neg x_i$ in the equation.

$$f_1(x) = \sum_{i=1}^{\tilde{d}} W_{(\mathbb{A},i)} \overline{x_i} \tag{6.6}$$

2. $f_2(x)$ is the sum of the WCETs of all basic blocks placed in the SPM, where $W_{(\mathbb{B},i)}$ represents the WCETs of the i^{th} basic block of the individual $\mathbb{B}$.

$$f_2(x) = \sum_{i=1}^{\tilde{d}} W_{(\mathbb{B},i)} x_i \tag{6.7}$$

3. $f_3(x)$ calculates the extra jump correction cost incurred when the succeeding basic block is in a different memory and has the allocation according to the individual $\mathbb{C}$.

$$f_3(x) = \sum_{i=1}^{\tilde{d}} (W_{(\mathbb{C},i)} - (W_{(\mathbb{A},i)} \overline{x_i} + W_{(\mathbb{B},i)} x_i)) \xi_{\mathbb{C}_i} \tag{6.8}$$

In Equation 6.8, we subtract the base WCET cost without jump correction of the i^{th} basic block, which yields the extra WCET cost required for jump correction. The base WCET cost that is subtracted ($W_{(\mathbb{A},i)}$ or $W_{(\mathbb{B},i)}$) depends on which memory the i^{th} basic block is allocated. To guarantee that we add the correct jump correction cost to the approximation model, we multiply this jump correction cost by $\xi_{\mathbb{C}_i}$.

$$\xi_{\mathbb{C}_i} = \begin{cases} 1, \text{ if } x_i = \mathbb{C}_i \ \& \ x_{i+1} = \mathbb{C}_{i+1}, \\ 0, \text{ otherwise} \end{cases} \tag{6.9}$$

where $\xi_{\mathbb{C}_i}$ is a d-dimensional binary vector, where 1s indicate that basic blocks b_i and b_{i+1} in the solution vector x have the same configuration as in the individual $\mathbb{C}$.

4. Similar to $f_3(x)$, $f_4(x)$ also calculates the extra jump correction cost incurred when the succeeding basic block is in a different memory. However, the allocation in this case is according to the individual $\mathbb{D}$.

$$f_4(x) = \sum_{i=1}^{\tilde{d}} (W_{(\mathbb{D},i)} - (W_{(\mathbb{A},i)}\overline{x_i} + W_{(\mathbb{B},i)}x_i))\xi_{\mathbb{D}_i} \tag{6.10}$$

Moreover, $\xi_{\mathbb{D}_i}$ is also a d-dimensional binary vector. However, the 1s in $\xi_{\mathbb{D}_i}$ indicate that basic blocks b_i and b_{i+1} in the solution vector x have the same configuration as in the individual $\mathbb{D}$, i.e.,

$$\xi_{\mathbb{D}_i} = \begin{cases} 1, \text{ if } x_i = \mathbb{D}_i \ \& \ x_{i+1} = \mathbb{D}_{i+1}, \\ 0, \text{ otherwise} \end{cases} \tag{6.11}$$

5. As mentioned before, the four individuals, $\mathbb{A}, \mathbb{B}, \mathbb{C}$, and $\mathbb{D}$, cannot cover all possible allocation cases. Therefore, $f_5(x)$ calculates the extra costs in the cases where the basic block branches to basic blocks other than its immediate successor or calls another function, and the current basic block and this second basic block are in different memories. Figure 6.2 shows example allocations that satisfy this scenario.

$$f_5(x) = \sum_{i=1}^{\tilde{d}} \mathbb{H}_i \vartheta_i \tag{6.12}$$

where ϑ is a d-dimensional binary vector. $\vartheta_i = 1$ if there exists an edge $e_{(i,k)} \in \mathbb{E}$ between the basic blocks b_i and b_k, the successor basic blocks

b_{i+1} and b_k are different, and basic blocks b_i and b_k are in different memories, i.e.,

$$\vartheta_i = \begin{cases} 1, \text{ if } b_k \neq b_{i+1} \;\&\; e_{(i,k)} \equiv (b_i, b_k) \in \mathbb{E} \;\&\; x_i \neq x_k, \\ 0, \text{otherwise} \end{cases} \tag{6.13}$$

Moreover, $\mathbb{H}_i$ represents the extra jump correction cost incurred due to the jump between basic blocks b_i and b_k across different memories, i.e.,

$$\mathbb{H}_i = \mathtt{WC}_i h_i \tag{6.14}$$

where $\mathtt{WC}_i$ represents the WCEC of basic block b_i obtained from the data collected during the initial WCET analysis, and h_i is the architecture-specific cost representing the extra cycles required to perform a single jump across different memories.

For ARM Cortex-M0, h_i is modeled as follows:

$$h_i = \begin{cases} 3 \text{ cycles, if } b_i \text{ has indirect call or explicit branch,} \\ 5 \text{ cycles, otherwise} \end{cases} \tag{6.15}$$

6. Similar to $f_5(x)$, $f_6(x)$ calculates the extra costs in the cases where the basic block branches to the basic blocks other than its immediate successor or calls another function. However, in this case, the current basic block and this second basic block are in the same memory. Figure 6.2 shows example allocations that satisfy this scenario.

$$f_6(x) = \sum_{i=1}^{\tilde{d}} \mathbb{L}_i \varphi_i \tag{6.16}$$

where φ is a d-dimensional binary vector and the basic blocks b_i and b_k are in the same memory, i.e.,

$$\varphi_i = \begin{cases} 1, \text{ if } b_k \neq b_{i+1} \;\&\; e_{(i,k)} \equiv (b_i, b_k) \in \mathbb{E} \;\&\; x_i = x_k, \\ 0, \text{otherwise} \end{cases} \tag{6.17}$$

Moreover, $\mathbb{L}_i$ is the extra jump correction cost incurred due to the jump between b_i and b_k within the same memory, i.e.,

$$\mathbb{L}_i = \mathtt{WC}_i l_i \tag{6.18}$$

where $\mathtt{WC}_i$ represents the WCEC of b_i, and l_i is the architecture-specific term representing the extra cycles required to perform a single jump within the same memory. During evaluations, we consider $l_i = 1$ cycle for ARM Cortex-M0.

7. $f_7(x)$ calculates the cost incurred while accessing the global data variables that are placed in the Flash.

$$f_7(x) = \sum_{g=(\tilde{d}+1)}^{d} \alpha T_F \theta_g \overline{x_g} \tag{6.19}$$

where αT_F is the access latency of the Flash and θ_g is the number of times the global data variable $\mathcal{G}_g$ is being accessed. During evaluations, we consider $\alpha T_F = 6$ cycles for ARM Cortex-M0.

8. $f_8(x)$ calculates the cost incurred while accessing the global data variables that are placed in the DSPM.

$$f_8(x) = \sum_{g=(\tilde{d}+1)}^{d} \alpha T_D \theta_g x_g \tag{6.20}$$

where αT_D is the access latency of DSPM. During evaluations, we consider $\alpha T_D = 1$ cycle for ARM Cortex-M0.

Using all eight terms defined above, we build the approximation model that approximates WCET as follows:

$$\begin{aligned} \widetilde{F_1}(x) = f_1(x) + f_2(x) + f_3(x) + f_4(x) \\ + f_5(x) + f_6(x) + f_7(x) + f_8(x) \end{aligned} \tag{6.21}$$

Equation 6.21 is the sum of all the costs associated with the basic blocks and global data objects within a program that are either placed in the SPM or the Flash memory.

6.3.3 Approximation Model for Energy Consumption

In this subsection, we propose the approximation model for the energy consumption objective. Let $E_{(f,i)}$ represent the energy consumption value of basic block $b_i \; \forall i = \overline{1, \tilde{d}}$ of the individual $f \in X$, and $\tilde{d}$ is the total number of basic blocks within a program. A strategy similar to WCET's approximation model

is employed to approximate the energy objective. Equation 6.22 represents the proposed model for approximating energy consumption.

$$\begin{aligned}\widetilde{F_2}(x) = &\sum_{i=1}^{\tilde{d}} E_{(\mathbb{A},i)}\overline{x_i} + \sum_{i=1}^{\tilde{d}} E_{(\mathbb{B},i)}x_i + \\ &\sum_{i=1}^{\tilde{d}} (E_{(\mathbb{C},i)} - (E_{(\mathbb{A},i)}\overline{x_i} + E_{(\mathbb{B},i)}x_i))\xi_{\mathbb{C}_i} + \\ &\sum_{i=1}^{\tilde{d}} (E_{(\mathbb{D},i)} - (E_{(\mathbb{A},i)}\overline{x_i} + E_{(\mathbb{B},i)}x_i))\xi_{\mathbb{D}_i} + \\ &\sum_{i=1}^{\tilde{d}} \mathcal{E}_{\mathbb{H}_i}\vartheta_i + \sum_{i=1}^{\tilde{d}} \mathcal{E}_{\mathbb{L}_i}\varphi_i + \\ &\sum_{g=(\tilde{d}+1)}^{d} \mathtt{AE}_F\theta_g\overline{x_g} + \sum_{g=(\tilde{d}+1)}^{d} \mathtt{AE}_S\theta_g x_g\end{aligned} \tag{6.22}$$

The purpose of each term in Equation 6.22 is similar to the terms in Equation 6.21. In this energy approximation model, instead of using WCETs ($W_{(f,i)}$), we use the energy values ($E_{(f,i)}$) of $\mathbb{A}, \mathbb{B}, \mathbb{C}$, and $\mathbb{D}$, where $f \in X$ and $i = \overline{1, d}$. The terms defined in Equations 6.9, 6.11, 6.13, and 6.17 are used in Equation 6.22 to approximate the energy objective. A small change occurs in the fifth and the sixth term, where $\mathcal{E}_{\mathbb{H}_i}$ and $\mathcal{E}_{\mathbb{L}_i}$ represent the extra energy cost incurred due to the jump between basic blocks b_i and b_k, i.e.,

$$\mathcal{E}_{\mathbb{H}_i} = \mathtt{AE}_F\mathbb{EC}_i\mathcal{J}_i \quad \text{and} \quad \mathcal{E}_{\mathbb{L}_i} = \mathtt{AE}_S\mathbb{EC}_i\mathcal{J}_i \tag{6.23}$$

where $\mathbb{EC}_i$ represents the execution count of the basic block b_i obtained from initial energy analysis, and the term $\mathcal{J}_i$ is an architecture-specific term representing the cost in terms of jump code size. $\mathcal{J}_i$ is the number of extra instructions added due to jump correction. For ARM Cortex-M0, $\mathcal{J}_i$ is modeled as follows.

$$\mathcal{J}_i = \begin{cases} 2, & \text{if } b_i \text{ has indirect call or explicit branch,} \\ 3, & \text{otherwise} \end{cases} \tag{6.24}$$

The jump code size cost $\mathcal{J}_i$ depends on the estimated number of extra instructions required to perform jump correction in different cases. If the last instruction of the basic block b_i is an indirect call or explicit branch, the extra cost in terms of code size incurred is a maximum of two instructions. In every other case, a maximum of three instructions is added.

Another small change occurs in the seventh and eighth terms, where $\mathtt{AE}_F$ is the architecture-specific term derived from the energy model of the respective

architecture, and it represents the energy cost to access data or instructions from the Flash memory. Similarly, $\mathtt{AE}_S$ is the architecture-specific term representing the energy cost required to access data or instructions from the SPM. For ARM Cortex-M0, $\mathtt{AE}_F = 2989$ and $\mathtt{AE}_S = 1441$ [WNNYE23, Tea20].

6.3.4 Approximation Model's Limitations

In this subsection, we discuss the limitations of the proposed approximation model. As mentioned in Section 6.3.1, the approximation model does not cover all possible scenarios we can encounter while performing static SPM allocation. Moreover, it does not take into consideration the execution context of a basic block.

If we consider a basic block that executes multiple times (for example, a basic block belonging to a loop), the basic block might have different execution times for each execution. This is generally referred to as the execution context of that basic block. The execution context of a basic block is used to distinguish between each execution of that basic block.

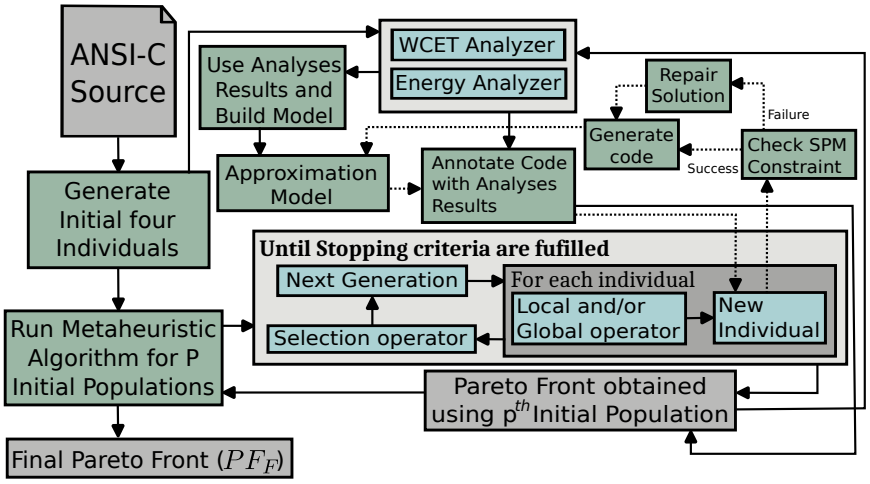
The results from the WCET and energy analyzers are affected based on the so-called unroll factor set during the analyses. The analyzer distinguishes between various loop iterations, i.e., the execution contexts of basic blocks, depending on the unroll factor. The WCET and energy consumption of each basic block vary depending on how many execution contexts of that basic block are considered. Therefore, the higher the unroll factor, the more accurate the WCET and energy analyzer can become.

However, as the proposed approximation model does not distinguish between the basic block's execution contexts, it might lead to some errors while approximating WCET and energy consumption. Therefore, approximation of WCET due to Equations 6.12, 6.16, 6.19, and 6.20 might lead to some approximation errors; similar errors can occur while approximating energy consumption. In Subsection 6.5.4, we evaluate individuals using both the approximation models and the WCET and energy analyzers to calculate such approximation errors.

6.4 Multi-Objective Optimization

In this section, we exploit the metaheuristic algorithms to perform and solve $\mathrm{MO}_{\mathrm{Approx}}$. Figure 6.3 shows the multi-objective optimization process and the structure of the metaheuristic algorithm library within WCC. We build the approximation model proposed in the previous section before running the metaheuristic algorithm. We only use the approximation model to approximate the objectives while running the metaheuristic algorithm. We represent these steps of generating statically allocatable code, calling the approximation model, and annotating the code with analysis results within the figure with dotted lines. However, we evaluate the Pareto front obtained at the end of

Figure 6.3.: Multi-objective static SPM allocation with the approximation model for approximating WCET and energy objectives



each optimization run (PF_F) using the WCET and energy analyzers to avoid introducing any approximation errors in the final results.

Algorithm 6.1 describes the pseudo-code for solving the MO_{Approx} problem. We define the optimization problem using the search space (X), the objective space (Y), the optimization constraints (cf. Equations 6.4 and 6.5), and the minimization objectives ($\widetilde{F}(x)$) (Line 1). We generate the four individuals—$\mathbb{A}, \mathbb{B}, \mathbb{C}$, and $\mathbb{D}$—as described in Section 6.3. We evaluate these individuals using the WCET and energy analyzers integrated within WCC, and then we collect the related data for these individuals (Line 2). We build the approximation model with this data and approximate the WCET and energy objectives using Equations 6.21 and 6.22 (Line 3).

Similar to the MO_S problem from Chapter 5, we use metaheuristic algorithms—FPA and SPEA—to solve the MO_{Approx} problem. We perform the evaluations using P initial populations and run the metaheuristic algorithm P times to solve the MO_{Approx} problem for different initial populations (Lines 4-27). Most of the steps from Line 4 to Line 27 within Algorithm 6.1 are similar to Algorithm 5.1 from Chapter 5, except Line 23. In Algorithm 6.1, we use the proposed approximation models defined by Equations 6.21 and 6.22 to evaluate the program's WCET and energy consumption instead of the WCET and energy analyzers. The approximation models are only used to guide the metaheuristic algorithms, and the final obtained solutions are analyzed using the WCET and energy analyzers for safety purposes.

6.5 Evaluation

In this section, we perform evaluations to compare MO_{Approx}, MO_S, and SO_S in terms of compilation times and the quality of obtained solutions. MO_{Approx} and MO_S are solved using two metaheuristic algorithms—FPA and

Algorithm 6.1 Multi-objective static SPM allocation with approximation model (MO_{Approx})

1: Define the MO_{Approx} problem (search space, objectives, and SPM size constraint)
2: Generate and evaluate individuals $\mathbb{A}, \mathbb{B}, \mathbb{C}$, and $\mathbb{D}$
3: Build an approximation model using the four individuals' analysis results
4: **for** $p = 1 : P$ **do** ▷ Run algorithm for P different initial populations
5: Randomly initialize the initial population of size N
6: **for** $n = 1 : N$ **do** ▷ Iterate over each individual in the initial population
7: Assign basic blocks to ISPM and global data variables to DSPM according to the initial individual vector
8: **while** $Q_1(x_{1:\tilde{d}}) > 0$ **do** ▷ Check the ISPM size constraint
9: Repair the solution by removing a basic block from ISPM
10: **while** $Q_2(x_{(\tilde{d}+1):d}) > 0$ **do** ▷ Check the DSPM size constraint
11: Repair the solution by removing a data variable from DSPM
12: Perform jump correction for the initial individual
13: Evaluate the initial individual using Equations 6.21 and 6.22
14: Initialize hyperparameters for the metaheuristic algorithms
15: **while** Stopping criteria are not reached **do** ▷ Iterate until the stopping criteria are fulfilled
16: **for** $n = 1 : N$ **do** ▷ Iterate over each individual in the current population
17: Update the individual using a global or local operator or both
18: **while** $Q_1(x_{1:\tilde{d}}) > 0$ **do** ▷ Check the ISPM size constraint
19: Repair the solution by removing a basic block from ISPM
20: **while** $Q_2(x_{(\tilde{d}+1):d}) > 0$ **do** ▷ Check the DSPM size constraint
21: Repair the solution by removing a data variable from DSPM
22: Perform jump correction
23: Evaluate the individuals using Equations 6.21 and 6.22
24: Update to the next generation using the selection operator
25: Evaluate the final generation using WCET and energy analyzers
26: Collect the Pareto optimal solutions obtained for p^{th} initial population
27: **Return:** Final Pareto optimal solution set (PF_F)

SPEA. Similar to Chapter 5, SO_S is an ILP-based single-objective static SPM allocation optimization. We also compare these approaches with a baseline (BL), where the whole code is executed from the Flash memory. Moreover, we analyze the approximation errors due to the estimations done in the proposed approximation model by comparing them to the results from the static WCET and energy analyzers.

Subsection 6.5.1 describes the evaluation setup used to solve MO_{Approx}, MO_S, and SO_S. Subsection 6.5.2 presents the compilation time results for

the MO_{Approx}, MO_S, and SO_S approaches. Subsection 6.5.3 discusses the Pareto optimal solutions found after solving MO_{Approx}, MO_S, SO_S, and BL evaluation runs. Subsection 6.5.4 presents the approximation error results for the proposed MO_{Approx} approach. Subsection 6.5.5 compares the quality of MO_{Approx}, MO_S, SO_S, and BL solutions using the quality indicators described in Subsection 4.3.3.

6.5.1 Evaluation Setup

The evaluations are conducted using the WCC compiler for the ARM Cortex-M0 architecture, similar to the approach in Chapter 5. The static WCET and energy analyses for MO_S,SO_S, and BL are performed using `aiT` v21.04i and `EnergyAnalyzer` v21.04i, respectively. The initial four individuals in MO_{Approx} are analyzed using the WCET and energy analyzers to build the approximation model. The rest of the optimization run uses the proposed approximation model, and the obtained final Pareto solutions are again safely analyzed using the WCET and energy analyzers. SO_S uses the ILP solver `Gurobi` [Gur24] integrated within WCC to solve the ILP problem.

We use the MRTC benchmark suite [GBEL10] with loop bound annotations from the TACLeBench project [FAH+16] for evaluation purposes. As done in Chapter 5, we compiled, analyzed, and evaluated 28 benchmarks. We generate the code using the standard `-O2` optimization flag within WCC. We adjust the SPM size individually to 60% relative to the benchmark size to increase the pressure on the optimization even for smaller benchmarks.

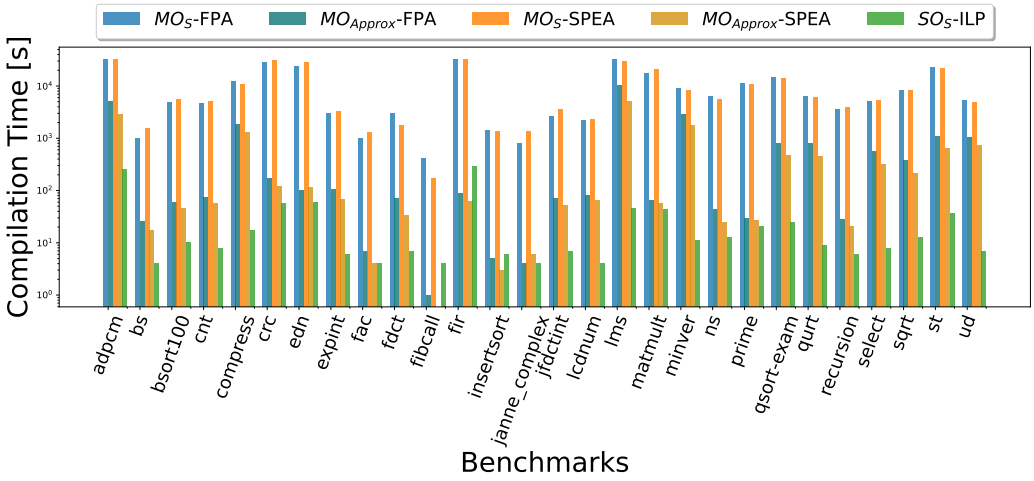
The initial population of any metaheuristic algorithm is defined randomly and can influence the final obtained solutions. Therefore, we repeat the evaluations for both FPA and SPEA using $P = 5$ different initial populations to reduce the influence of the initial population on the final results. In the end, we combine all the individual Pareto fronts (PF_F) from all the optimization runs to find the final Pareto front (PF_T). We set a timeout value of 20 h for each benchmark, and the compiler terminates the compilation if the optimizations have not finished successfully. If the optimization is not complete within the time limit, we derive a Pareto front from the current running generation and terminate the optimization.

To solve the MO_S and MO_{Approx} optimizations using FPA and SPEA algorithms, we use the hyperparameter values described in Tables 5.1 and 5.2 in Chapter 5. The results obtained during these evaluations are valid for the algorithm parameters described in these tables.

6.5.2 Compilation Times

In this subsection, we compare the compilation time required by MO_{Approx}, MO_S, and SO_S. As seen in Chapter 5, the MO_S optimization is time-consuming compared to SO_S due to multiple time-consuming WCET and energy analyses required to finish the optimization. Therefore, in this chapter, we proposed

Figure 6.4.: Compilation times for MO_{Approx}, MO_S, and SO_S optimizations



the approximation model that replaced the time-consuming analyses with WCET and energy approximations while solving the MO_{Approx} optimization.

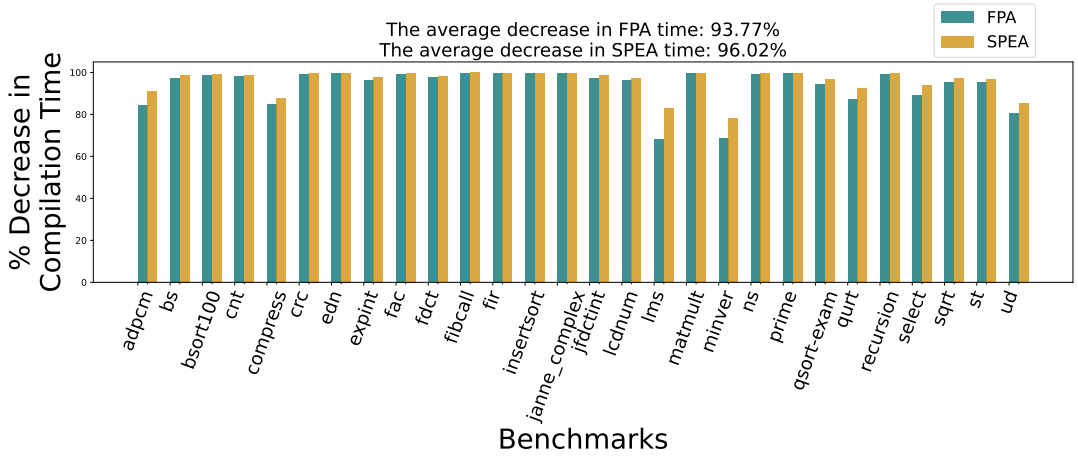
Figure 6.4 shows the comparison of the compilation time required by MO_{Approx}–FPA, MO_{Approx}–SPEA, MO_S–FPA, MO_S–SPEA, and SO_S–ILP optimizations in the form of a bar plot. The x-axis represents all the evaluated benchmarks, and the y-axis shows the compilation time required to complete the optimization runs. For brevity's sake, we are not comparing the compilation time required by BL in Figure 6.4.

From the figure, we can see that MO_{Approx} is faster than the time-consuming MO_S. MO_{Approx}–SPEA took 8.78 min, and MO_{Approx}–FPA took 15.38 min, on average, to finish their respective optimization runs. MO_S–FPA and MO_S–SPEA took 2.95 h and 3 h, on average, to finish the optimization runs, respectively. However, SO_S is still faster than the MO_{Approx} optimization runs. As mentioned in Chapter 5, SO_S took 0.59 min to finish its optimization runs.

MO_{Approx} performs four initial WCET and energy analyses using respective analyzers, and it performs WCET and energy analyses of the final Pareto front using the analyzers. The size of the final Pareto front is, at most, equal to the size of the population—we set the size of the population to 10 (cf. Tables 5.1 and 5.2). Moreover, we perform the evaluations for $P = 5$ different initial populations, which, in turn, can produce five final Pareto fronts. Therefore, for each benchmark, MO_{Approx}, at most, performs $(5 * 2 * 10 + 4 * 2) = 108$ total WCET and energy analyses using respective analyzers. This number can decrease significantly depending on the size of the final Pareto front. Therefore, compared to SO_S, MO_{Approx} is a bit more time-consuming.

We further compare the reduction in compilation times (in %) between MO_{Approx} and MO_S for each benchmark using Figure 6.5. The x-axis of the figure shows the evaluated benchmarks from the MRTC benchmark suite, and the y-axis represents the percentage decrease in the compilation times achieved

Figure 6.5.: Percentage decrease in the compilation times achieved by using approximation-based multi-objective static SPM allocation



using MO_{Approx} compared to MO_S. The legend at the top-right corner of the figure represents the metaheuristics used during evaluation, i.e., FPA and SPEA. Each bar for each benchmark in Figure 6.5 represents the percentage decrease in the compilation times achieved using MO_{Approx} over MO_S. Using the FPA algorithm, on average, MO_{Approx} achieved a 93.77% decrease in compilation times compared to MO_S. Using the SPEA algorithm, on average, MO_{Approx} achieved a 96.02% decrease in compilation times compared to MO_S.

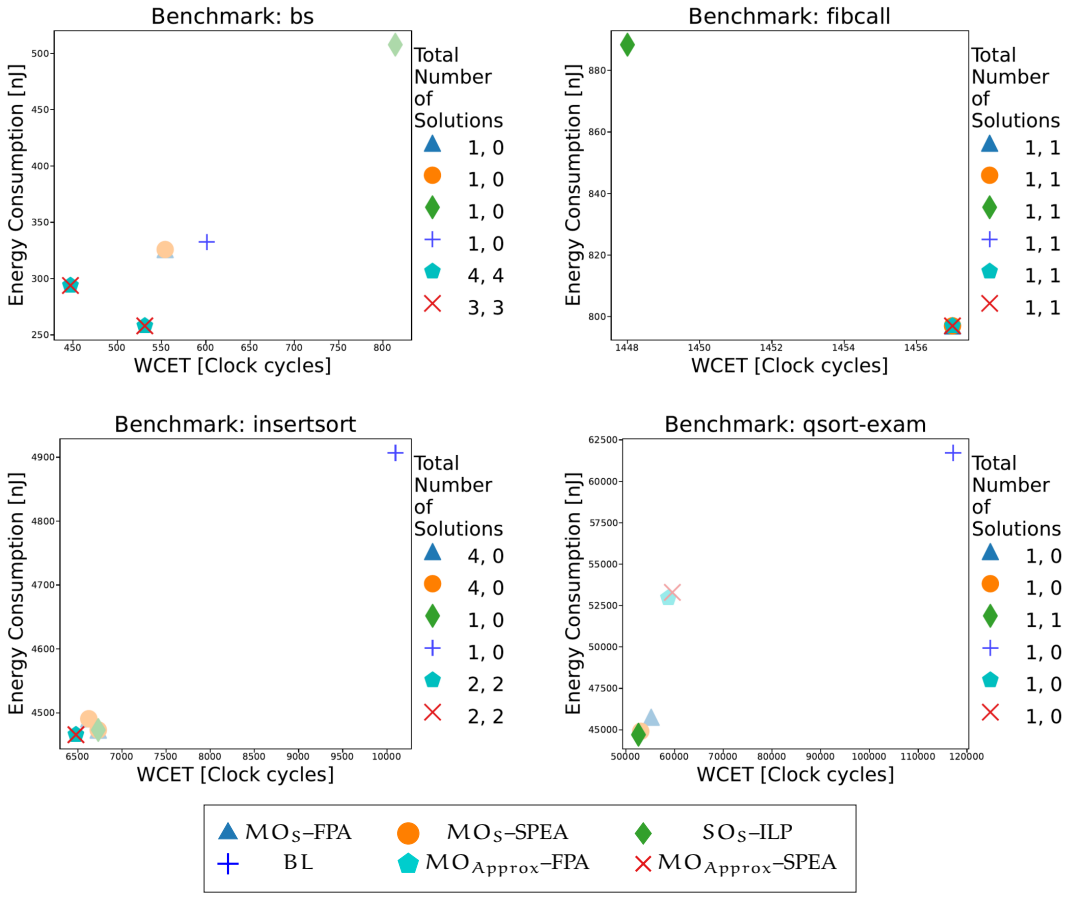
Finally, we achieved a significant percentage decrease in compilation times of 94.89% on average by using the proposed approximation model in conjunction with FPA and SPEA algorithms over the traditional WCET and energy analyzers while performing multi-objective optimization. Furthermore, we calculated the sum of all the compilation times for all the benchmarks, and the total time taken by MO_S–FPA and MO_S–SPEA optimization runs are 82.69 h and 84.26 h, respectively, while MO_{Approx}–FPA and MO_{Approx}–SPEA took just 7.17 h and 4.1 h, respectively.

6.5.3 Pareto Fronts

In this subsection, we compare all the solutions obtained after performing MO_{Approx}, MO_S, and SO_S optimizations. Moreover, we also perform a baseline BL evaluation and compare it with the optimization approaches. Figure 6.6 shows the solutions found by FPA and SPEA using both MO_{Approx} and MO_S. It also shows the solutions contributed by the SO_S–ILP optimization and the BL evaluation run. For the sake of brevity, this subsection presents the solutions for four benchmarks (similar to the benchmarks in Figure 5.3). The figures for all the remaining benchmarks are in Appendix B.

In Figure 6.6, we represent the Pareto fronts using a 2D scatter plot for the following four benchmarks: `bs`, `fibcall`, `insertsort`, and `qsort-exam`. The

Figure 6.6.: Pareto fronts for four benchmarks using MO_S, SO_S, BL, and MO_{Approx}



x-axis and the y-axis represent WCET and energy objectives, respectively. The title of each plot is the benchmark name. The legend at the bottom of the figure represents the approach to which a particular solution belongs. The legend on the right side of each plot shows the total number of solutions returned by the respective approach on their final Pareto fronts (PF_F). Moreover, the same legend presents the number of solutions out of all PF_Fs on the combined final Pareto front (PF_T). The final Pareto front (PF_T) represents the set of indifferent solutions from the union of the Pareto optimal solutions obtained from all the approaches. In the figure, the darker colors represent the solutions on PF_T, and the fainter versions of those colors represent the solutions returned by each approach on their Pareto fronts (PF_F). As we considered a minimization problem, the solutions in the bottom-left corner of the plots generally represent the final Pareto front.

For `bs`, both MO_{Approx}–FPA and MO_{Approx}–SPEA provided solutions on the final PF_T. MO_{Approx} performed better than the rest of the approaches. In the case of the `fibcall` benchmark, we end up with the same result as

Table 6.1.: Comparison between the average number of solutions (in %) on the final PF_T from the respective PF_F of each optimization approach

MO_S		SO_S	BL	MO_{Approx}	
FPA	SPEA	ILP		FPA	SPEA
12.12	15.52	32.14	3.57	24.1	42.5

discussed in Chapter 5. The solutions obtained from MO_{Approx} and MO_S are the same as the BL solution, i.e., no optimization was performed.

For `insertsort`, MO_{Approx} performed better than the rest of the approaches; MO_{Approx}–FPA and MO_{Approx}–SPEA both provided two solutions each on the final PF_T. However, in the case of the `qsort-exam` benchmark, MO_{Approx} did not contribute any solution to PF_T; the only solution provided by SO_S formed the final PF_T.

From Figures 6.6, B.1, B.2, and B.3, we can see that: MO_S–FPA and MO_S–SPEA contributed solutions to the final PF_T for 5 and 3 benchmarks, respectively; SO_S contributed solutions to PF_T for 9 benchmarks; BL contributed to PF_T for 1 benchmark (the `fibcall` case from Figure 6.6); MO_{Approx}–FPA and MO_{Approx}–SPEA contributed solutions to the final PF_T for 9 and 16 benchmarks, respectively. Therefore, we can say that MO_{Approx}–SPEA performed better than MO_S, SO_S, and BL. MO_{Approx}–FPA and SO_S performed equally well.

From these evaluations, it seems like MO_S performed worse than SO_S. However, for the benchmarks where MO_S performed better than SO_S, MO_{Approx} outperformed MO_S and provided solutions on the final PF_T. Therefore, the results from Chapter 5—MO_S performed slightly better than SO_S—still hold. Moreover, along with being fast, MO_{Approx} performed better than MO_S in terms of providing solutions to the final PF_T.

From overall evaluations, we calculated the total number of solutions provided by each optimization run on the final PF_T and averaged it over all the benchmarks. Table 6.1 provides the percentage of the number of solutions on the final Pareto front PF_T, on average, out of the number of solutions on respective optimization's PF_F. From the table, we can see that MO_{Approx}–SPEA provided the largest number of solutions on the final PF_T. MO_{Approx}–FPA performed worse than MO_{Approx}–SPEA, but it still performed better than MO_S.

As indicated in Table 6.1, SO_S–ILP has a higher percentage of solutions on PF_T from its respective PF_Fs compared to MO_{Approx}–FPA. However, as mentioned in Subsection 5.4.2, this number can be skewed towards single-solution-producing approaches. Overall, SO_S–ILP provided 9 solutions on PF_T out of 28 solutions from its PF_Fs, and MO_{Approx}–FPA contributed 20 solutions to PF_T out of 80 solutions from its PF_Fs.

6.5.4 Approximation Errors

In this subsection, we perform an MO_{Approx} optimization run to calculate the approximation error using the percentage mean l_1 error. During this MO_{Approx} run, we evaluate each individual simultaneously using the approximation model and the respective analyzers to calculate the said error. Let $\nu = \{\nu^1, \ldots, \nu^N\}$ be the objective values evaluated using the analyzer, and $\widetilde{\nu} = \{\widetilde{\nu}^1, \ldots, \widetilde{\nu}^N\}$ be the objective values approximated by the proposed approximation model, where N is the total number of individuals over all the populations and generations of the algorithm. Then, the percentage mean l_1 error is defined as:

$$ME_p^{l_1} = \frac{1}{N} \sum_{q=1}^{N} \frac{|\widetilde{\nu}^q - \nu^q|}{\nu^q} * 100 \tag{6.25}$$

Using this equation, we calculate approximation errors for all benchmarks. The proposed approximation model for WCET returned, on average, 5.73% and 6.29% mean l_1 error over all the benchmarks for FPA and SPEA, respectively. The proposed approximation model for energy, on average, returned 4.95% and 5.4% mean l_1 error for FPA and SPEA, respectively.

The `lcdnum` benchmark returned a maximum approximation error of 17.56% for WCET, and `adpcm` returned a maximum approximation error of 13.95% for energy consumption. However, in the case of both benchmarks, only MO_{Approx} was able to contribute to the final PF_T, and it performed better than all other approaches (cf. Figures B.1 and B.2). Furthermore, the `fdct` benchmark returned a minimum approximation error of 0.17% for WCET, and the same benchmark returned a minimum approximation error of 0.13% for energy consumption.

As mentioned in Subsection 6.3.4, the execution context of each basic block is not considered while approximating WCET and energy consumption. This might lead to differences in the approximated and analyzed WCET and energy consumption values. The approximation model is more accurate when the similarity between the considered individual and the four initial individuals is higher. Moreover, the approximation model for WCET tries to compensate for the rest of the cases by using Equations 6.12, 6.16, 6.19, and 6.20; similar terms are part of the approximation model for energy consumption. However, they are not accurate in every possible case, leading to such approximation errors.

As mentioned before, the approximation model is only used to iteratively analyze the individuals within the populations generated during the run of the metaheuristic algorithm. The final Pareto optimal population set outputted by the metaheuristic algorithm is then evaluated using the WCET and energy analyzers integrated within WCC. The evaluation of the final Pareto optimal population using WCC's analyzers provides the user with properly analyzed binaries without minor approximation errors. Therefore, the approximation errors calculated during this evaluation run refer only to the evaluation of

Table 6.2.: Total number of benchmarks with best (or nondominated) quality indicators for each optimization approach

Quality Indicators	MO_S		SO_S	BL	MO_{Approx}	
	FPA	SPEA	ILP		FPA	SPEA
$\mathcal{C}$	2 (3)	2 (1)	7 (2)	0 (1)	0 (8)	7 (9)
NR	2 (2)	2 (2)	6 (3)	0 (1)	2 (7)	6 (8)
RNS	2 (3)	2 (1)	7 (2)	0 (1)	0 (8)	7 (9)
HV	3 (1)	2 (1)	6 (0)	0 (1)	6 (3)	8 (6)

individuals generated only during the metaheuristic algorithm run using the proposed approximation model. The quality and the quantity of the solutions obtained during these evaluations are evaluated and interpreted in detail using quality indicators in the next subsection.

6.5.5 Quality Indicators

Similar to Subsection 5.4.3, we use the following four quality indicators to further evaluate and compare the quality of the proposed MO_{Approx} approach: Coverage ($\mathcal{C}$), Non-dominance Ratio (NR), Ratio of Non-dominated Solutions (RNS), and Hypervolume (HV) (cf. Subsection 4.3.3). Moreover, under a realistic assumption that the exact Pareto front is unknown, we consider the final Pareto front PF_T discussed in Subsection 6.5.3 as the best available approximation of the exact Pareto front to calculate the quality indicators.

Table 6.2 summarizes the total number of benchmarks with best or nondominated $\mathcal{C}$, NR, RNS, and HV for MO_S, SO_S, BL, and MO_{Approx}. The notion for Table 6.2 is similar to Table 5.5 from Chapter 5. In Table 6.2, the numbers shown inside the brackets represent the number of benchmarks with nondominated Pareto optimal solutions compared to the other approaches, and the numbers outside the brackets represent the total number of benchmarks with the best quality indicators. For reference sake, Tables B.1 and B.2 in Appendix B present the quality indicator values for all the evaluated benchmarks from these evaluations.

From Table 6.2, we can see that MO_{Approx} performed better than MO_S for a larger number of benchmarks. SO_S–ILP and MO_{Approx}–FPA performed equally well in terms of the number of benchmarks that provided solutions to the final PF_T. Lastly, we can see that MO_{Approx}–SPEA performed better than all other approaches.

From Table 6.2, it seems like MO_S is somehow performing worse than SO_S. However, if we look at Table 5.5, MO_S performed slightly better than SO_S. MO_{Approx} provided better solutions to the benchmarks where MO_S was previously performing better than SO_S. Therefore, we cannot make a direct comparison between MO_S and SO_S using Table 6.2.

Due to the random nature of metaheuristic algorithms, we cannot conclude that MO_{Approx} will always converge to better solutions compared to MO_S.

However, during these particular evaluations, MO_{Approx} definitely outperformed MO_S. Moreover, given that MO_{Approx} achieved a 94.89% decrease in the compilation time on average, we can say that MO_{Approx} is a better option for performing a fast multi-objective optimization compared to MO_S. Therefore, we see a clear merit in using MO_{Approx} to perform multi-objective optimization over MO_S.

6.6 Conclusion

In this chapter, we proposed a novel approximation model that approximates WCET and energy consumption at compile-time. We also combined the SPM allocation-based multi-objective optimization with the approximation model to perform objective evaluations. We evaluated the MO_{Approx} optimization problem using two metaheuristic algorithms—FPA and SPEA. We compared the results from MO_{Approx}, MO_S, SO_S, and BL evaluation runs to evaluate the efficiency of the proposed approximation model.

The approximation errors during the MO_{Approx} optimization run were in the acceptable range of 4% - 7% on average. Using MO_{Approx}, we achieved, on average, a 94.89% decrease in compilation time compared to MO_S. MO_{Approx}–FPA performed comparable to SO_S, providing solutions on the final PF_T for the same number of benchmarks. However, MO_{Approx}–FPA provided more total solutions to PF_T compared to SO_S. MO_{Approx}–SPEA performed better than all the other approaches in all aspects. Therefore, we can conclusively say that using the proposed approximation model can drastically reduce the required compilation time and achieve good quality Pareto optimal solutions.

Performing multi-objective optimization at the compiler level requires a large amount of effort in terms of compilation times. Dealing with expensive optimization objectives like WCET and energy consumption contributes immensely to this problem. Therefore, the approximation model proposed in this chapter helps to deal with one of the biggest bottlenecks from Chapter 5 while performing compiler-level multi-objective optimization.

The approximation model proposed in this chapter depends on the underlying compiler optimization problem, i.e., static SPM allocation, and the underlying architecture-specific details. In the next chapter, instead of approximating objectives, we aim to reduce the search space dimension of the optimization problem by exploiting the objective function's properties. The approach proposed in the next chapter does not rely on the underlying compiler optimization or the architecture-specifics, but on the multi-objective optimization's objective.

7

Search Space Reduction of Multi-Objective Static SPM Allocation for Multi-Task Systems

In Chapter 5, we explored multi-objective static SPM allocation optimization and solved it using two metaheuristic algorithms—FPA and SPEA. In Chapter 6, we proposed an approximation model to approximate WCET and energy objectives to deal with the high compilation time required to solve the multi-objective optimization problem proposed in Chapter 5. However, the approximation model proposed in Chapter 6 has the following limitations:

- The approximation model depends on the underlying compiler-level optimization, i.e., we need to change the approximation model to use it to approximate WCET and energy consumption for any compiler optimization other than SPM allocation.

- The approximation model consists of architecture-specific constants needed due to jump correction costs. Therefore, we need to modify the approximation model to accommodate these architecture-specific details.

Each compiler-level optimization is different, dealing with various constraints and conditions. For example, if the underlying compiler optimization is not defined at the basic block level, we cannot use the proposed approximation model. Moreover, there are so many variations in embedded architectures that it is impossible to deal with all possible architectures without cluttering the model. However, the approximation model proposed in Chapter 6 is generic enough that by changing the values of architecture-specific constants, we could try to accommodate various architectures.

In this chapter, we propose two approaches—Path-based Constraint Approach (PCA) and Impact-based Constraint Approach (ICA)—that aim to reduce the problem's search space to reduce the compilation time required to solve the multi-objective static SPM allocation problem. The PCA and ICA techniques depend on the underlying objective of the multi-objective optimization. For example, the WCET of a program depends on its WCEP (cf. Definition 3.4), so the PCA technique constrains all basic blocks that are not

on the WCEP, reducing the problem's search space. The following points can be made regarding the proposed PCA and ICA techniques:

- The proposed approaches need basic block-level objective information to reduce the size of the search space.

- They do not rely on the approximation of the objectives. Therefore, we need to perform the analyses of each individual using the respective analyzers.

- They do not rely on the underlying compiler optimization. If the search space of the optimization is defined by the program's basic blocks, we can use the PCA and ICA techniques.

In Chapters 5 and 6, we proposed and solved a multi-objective static SPM allocation problem for a single-task system that aims to minimize two objectives simultaneously—WCET and energy consumption. In this chapter, we extend the multi-objective static SPM allocation problem to handle multi-task systems, and we consider schedulability as a minimization objective along with WCET and energy consumption. For brevity's sake, we hereafter refer to the multi-objective static SPM allocation optimization for multi-task systems as MTMO.

Traditionally, the schedulability of a multi-task system is considered as a constraint, i.e., if the considered solution is not schedulable, the optimization will reject that solution. However, in this chapter, we do not discard such solutions by treating schedulability as an objective; the lower the objective value, the more tasks in the multi-task systems are schedulable. It is important to note that if we consider schedulability as a constraint, depending on the metaheuristic algorithm, it can either discard that solution and generate a new individual randomly or try to repair that solution.

- If the metaheuristic algorithm's policy is to discard the solution, there is no guarantee that the newly randomly generated individual would be better than the current discarded individual.

- If the metaheuristic algorithm opts to repair the solution, we need to implement a repair mechanism that tries to repair the solution such that it becomes schedulable. However, we are solving the optimization problem to find such solutions—we have encountered a cyclical issue.

By considering schedulability as an objective, we try to keep the unscheduled solutions in contention, and the metaheuristic algorithm's update operators will update the individuals themselves. Therefore, by iterating over each population, the metaheuristic algorithm will try to find solutions with better schedulability objectives over each generation.

The proposed MTMO optimization for multi-task systems explores the complete search space. However, the proposed search space reduction techniques—PCA and ICA—solve the MTMO problem, where we explore the reduced search space.

The MTMO, PCA, and ICA techniques described in this chapter were presented at the International Workshop on Worst-Case Execution Time Analysis (WCET) in 2023 [JF23a]. The key contributions of this chapter are:

- We formulate static SPM allocation as a multi-objective optimization problem for a multi-task system (MTMO), considering schedulability, WCET, and energy consumption as minimization objectives.
- After solving the problem, WCC returns a set of Pareto optimal solutions that trade off the three objectives.
- We proposed two approaches—PCA and ICA—to reduce the search space size to cope with the size of the optimization problem.

During evaluations, we compare the MTMO results with the PCA and ICA techniques in terms of solution quality and required compilation times. We also compare the proposed MTMO, PCA, and ICA approaches with a baseline (BL) evaluation, where the whole multi-task task set is executed from the Flash memory. The key contributions of this chapter are:

- PCA and ICA optimization runs reduced compilation times by 23.93% and 24.66%, on average, over the MTMO optimization run.
- PCA and ICA achieved, on average, 23.97% and 52.83% reduction in search space compared to MTMO.
- ICA–FPA performed better than other approaches in terms of solution quality and compilation times.

This chapter is organized as follows: Section 7.1 provides an overview of the related background work. Section 7.2 defines the MTMO problem for static SPM allocation. Section 7.3 introduces the MTMO framework. Sections 7.4 and 7.5 propose the PCA and ICA techniques to reduce the search space size of the MTMO optimization problem, respectively. Section 7.6 presents the evaluation results, and Section 7.7 concludes the chapter.

7.1 Related Work

A multi-task system is called *schedulable* if it meets all its timing constraints, even under worst-case circumstances (cf. Definition 2.8). A multi-task system, which is not schedulable in its initial state, can be optimized in various ways to obtain a schedulable system. For example, we can use optimal scheduling algorithms [LL73, Leh90, Bar03] to find schedulable system configurations,

increase hardware capabilities to optimize the runtime behavior, etc. Increasing hardware capabilities, unfortunately, is not only expensive but also usually comes with higher energy consumption. Therefore, we can exploit compiler optimizations to improve the worst-case behavior of the system, inadvertently fulfilling all the timing constraints of the system.

Zhao et al. [ZKW+06] proposed an approach to reduce the program's WCET by adapting and applying optimizations designed for average-case paths to the program's WCEPs. Luppold et al. [LF15, LF17] proposed compiler optimizations that perform ILP-based schedulability-aware static instruction SPM allocation for strictly periodic and arbitrarily triggered multi-task systems, respectively. Moreover, Luppold et al. [LOF20] also presented a compiler optimization technique for a multi-task and multi-core real-time system, which uses an ILP model to perform schedulability-aware optimization focusing on the minimization of the WCET of task sets.

Xian et al. [XLL07] proposed an energy-aware scheduling for multiprocessor systems. They considered probabilistic distributions of the execution times of multiple tasks to partition the workload for better energy reduction. Takase et al. [TZG+11] proposed a framework for reducing the energy consumption of multi-task systems during runtime by trading energy and performance of the processor configuration and selecting the optimal processor configuration at each appropriate point in the task.

The schedulability of a multi-task system has been exploited primarily to optimize the runtime behavior of a system. Furthermore, schedulability-aware compile-time techniques focus on the minimization of a single objective. As a result, while optimizing a system on a code level, a system designer currently has to choose whether to minimize energy consumption, which might lead to a non-schedulable system, or whether to use schedulability-oriented optimization techniques, which might drive energy consumption needlessly high. Therefore, in this chapter, we formulate a multi-objective optimization problem that simultaneously considers WCET, schedulability, and energy consumption as minimization objectives.

7.2 Problem Formulation

In this section, we formulate a 3-dimensional multi-objective static SPM allocation optimization problem for a multi-task system that aims to minimize WCET, energy consumption, and schedulability simultaneously. Static SPM allocation is a compiler-level optimization where we statically move basic blocks from the slow Flash memory to the fast and energy-efficient SPM. In this problem formulation, we treat schedulability as an objective to minimize rather than a constraint. As mentioned before, if we consider schedulability as a constraint, the metaheuristic algorithm might reject the solutions violating this constraint completely, or we might need to repair such solutions, creating

a cyclical problem. Therefore, we treat schedulability as an objective while solving the proposed MTMO problem.

As the schedulability objective depends on the task's WCET, we could consider only schedulability as a minimization objective, but it might hinder the exploration of the search space. For example, a multi-task system with a certain WCET value can be schedulable. However, more schedulable solutions with lower WCET values could exist within the search space. Lastly, WCET and energy consumption objectives can contradict each other. Therefore, we simultaneously consider WCET, energy consumption, and schedulability as minimization objectives. As a result, the Pareto optimal solutions obtained by solving this problem will lie in a 3-dimensional objective space.

As mentioned and defined in Sections 2.3 and 3.3, for a multi-task system, $\tau_t \in \Gamma$ is the t^{th} task in the set of all tasks Γ. The multi-objective static SPM allocation optimization problem for a multi-task system (MTMO) is mathematically formulated as follows:

$$\begin{aligned} \min_{x} \quad & F(x) = (F_1(x), F_2(x), F_3(x)) \\ \text{subject to} \quad & Q_1(x) \leqslant 0 \text{ and } Q_2(x) \leqslant 0 \end{aligned} \tag{7.1}$$

where $x = (x_1, \ldots, x_T) \in X \subset \{0,1\}^d$ is a d-dimensional binary decision vector for the multi-task set Γ. x_t is a binary decision subvector for a single task $\tau_t \in \Gamma$, where $t = \overline{1,T}$, and T is the total number of tasks in the multi-task set Γ, i.e., $T = |\Gamma|$. The subvector x_t is further subdivided into two subvectors: the first is the decision subvector for basic blocks ($x_{t_{1:\tilde{d}_t}}$), and the second is the decision subvector for global data variables ($x_{t_{(\tilde{d}_t+1):d_t}}$).

For static SPM allocation purposes, the decision of placing an i^{th} basic block from the t^{th} task (b_{t_i}) in ISPM or Flash is realized by a decision variable $x_{t_i} \in \{0,1\}$ of the decision vector, i.e.,

$$\forall i = \overline{1,\tilde{d}_t} \quad x_{t_i} = \begin{cases} 1, & \text{if } b_{t_i} \text{ is placed in ISPM,} \\ 0, & \text{otherwise} \end{cases} \tag{7.2}$$

The subvector $x_{t_{(\tilde{d}_t+1):d_t}}$ decides whether global data variables ($\mathcal{G}_{t_g}$) will be placed in DSPM or Flash, i.e.,

$$\forall g = \overline{\tilde{d}_t+1,d_t} \quad x_{t_g} = \begin{cases} 1, & \text{if } \mathcal{G}_{t_g} \text{ is placed in DSPM,} \\ 0, & \text{otherwise} \end{cases} \tag{7.3}$$

$\tilde{d}_t$ represents the total number of basic blocks in the t^{th} task, $(d_t - \tilde{d}_t)$ is the total number of global data variables in the t^{th} task, and $d = \sum_{t=1}^{T} d_t$ is the total size of the binary decision variable vector.

The first constraint condition, $Q_1(x) \leqslant 0$, for the MTMO problem represents the number of basic blocks that can fit in the ISPM statically.

$$Q_1(x) = \sum_{t=1}^{T} \sum_{i=1}^{\tilde{d}_t} B_{t_i} x_{t_i} + \Psi_J - S_{ISPM} \tag{7.4}$$

where B_{t_i} is the code size of the basic block b_{t_i}, b_{t_i} is the i^{th} basic block in the t^{th} task, S_{ISPM} represents the ISPM size, and Ψ_J represents the extra code size cost incurred due to jump correction. Jump correction and the jump correction cost Ψ_J are discussed in detail in Sections 3.1.3 and 5.2, respectively.

The second constraint condition, $Q_2(x) \leqslant 0$ in Equation 7.1, represents the number of global data variables that can fit in the DSPM statically.

$$Q_2(x) = \sum_{t=1}^{T} \sum_{g=(\tilde{d}_t+1)}^{d_t} S_{\mathcal{G}_{t_g}} x_{t_g} - S_{DSPM} \tag{7.5}$$

where $S_{\mathcal{G}_{t_g}}$ is the size of the global data variable $\mathcal{G}_{t_g}$, and S_{DSPM} is the total size of the DSPM.

$F(x) = (F_1(x), F_2(x), F_3(x))$ is a 3-dimensional objective function that maps the decision vector $x \in X$ from the search space $X \subset \{0,1\}^d$ to the 3-dimensional objective space $Y \subset \mathbb{R}^3$. $F_1(x)$ and $F_2(x)$ are the total WCET and energy objectives for the binary decision vector x, i.e.,

$$F_1(x) = \sum_{t=1}^{T} W_t \quad \text{and} \quad F_2(x) = \sum_{t=1}^{T} E_t \tag{7.6}$$

where W_t and E_t are WCET and energy consumption of task $\tau_t \in \Gamma$, respectively. After performing WCET and energy analysis, these respective objective values $F_1(x)$ and $F_2(x)$ are collected within WCC.

Often, the schedulability of a system is considered as a constraint, that is to say, whether the system is schedulable or not. However, as mentioned before, in our problem, schedulability is treated as a minimization objective, with the best case where all the tasks are schedulable and in the worst case, none are schedulable.

To consider schedulability as an objective, we use the schedulability analyzer integrated within WCC to calculate the number of tasks that might have to be removed from the system to achieve schedulability. If a system with T tasks is schedulable and all the tasks safely meet their corresponding deadlines, the analyzer will return 0. On the other hand, in the worst-case scenario, where no task safely meets its corresponding deadline, the analysis will return T. The schedulability analyzer integrated within WCC is discussed briefly in Section 3.3. $F_3(x)$ in Equation 7.1 is the schedulability objective, and it is formulated as follows:

$$F_3(x) = \sum_{t=1}^{T} \neg \nu_t W_t \tag{7.7}$$

where W_t is the WCET of the t^{th} task in the multi-task set Γ and ν_t is the binary ILP variable from Equation 3.2. The ILP variable $\nu_t \equiv 0$ indicates that the task τ_t is a task removed from Γ, i.e., $\tau_t \in \Gamma_r$, where Γ_r denotes a set of tasks removed from Γ to make the system of remaining tasks schedulable. Therefore, we use $\neg\nu_t$ in Equation 7.7 to add the WCET value of the removed task to the schedulability objective.

We use the ILP-based schedulability analyzer (cf. Section 3.3) within WCC to calculate the number of tasks needed to be removed from the system to achieve schedulability, i.e., if the schedulability analyzer returns an empty Γ_r set, the system is schedulable without removing any tasks. If all the tasks from Γ are in the returned Γ_r, we have encountered the worst-case scenario.

The final Pareto optimal solution set returned by the multi-task multi-objective static SPM allocation optimization could consist of solutions that are either fully or partially schedulable and indifferent in terms of WCET and energy consumption. Therefore, after solving the proposed MTMO optimization, the system designer will have a set of Pareto optimal binaries. Depending on the system under consideration, the system designer can choose a suitable solution from the set of Pareto optimal binaries based on the runtime requirements.

7.3 Multi-Task Multi-Objective Static SPM Allocation Optimization

In this section, we exploit the metaheuristic algorithms to perform and solve the compiler-level multi-objective static SPM allocation optimization (MTMO) problem for multi-task systems. The MTMO problem aims to simultaneously minimize three objectives—WCET, energy consumption, and schedulability. As the MTMO problem is a multi-task extension of MO_S from Chapter 5 and considers an extra schedulability constraint than MO_S, the steps required to solve the MTMO problem look similar to the MO_S problem described in Section 5.3.

Algorithm 7.1 describes the pseudo-code for solving the MTMO problem. We define the MTMO optimization problem using the search space (X), the objective space (Y), the SPM size constraints (cf. Equations 7.4 and 7.5), and the minimization objectives ($F(x)$) (Line 1). We perform the evaluations using a metaheuristic algorithm for P initial populations to reduce the influence of the initial population on the final solutions (Line 2). Similar to Algorithm 5.1, we randomly initialize the initial population and check if it satisfies the ISPM and DSPM size constraints (Lines 3-5). If the SPM size constraints are not satisfied,

Algorithm 7.1 Multi-task multi-objective static SPM allocation (MTMO)

1: Define the MTMO optimization problem (search space, objectives, and SPM size constraint)
2: **for** $p = 1 : P$ **do** ▷ Run algorithm for P different initial populations
3: Randomly initialize the initial population of size N
4: **for** $n = 1 : N$ **do** ▷ Iterate over each individual in the initial population
5: Assign basic blocks to ISPM and global data variables to DSPM according to the initial individual vector
6: **while** $Q_1(x) > 0$ **do** ▷ Check the ISPM size constraint
7: Repair the solution by removing a basic block from ISPM
8: **while** $Q_2(x) > 0$ **do** ▷ Check the DSPM size constraint
9: Repair the solution by removing a data variable from DSPM
10: Perform jump correction for the initial individual
11: Perform WCET, energy, and schedulability analyses
12: Initialize hyperparameters for the metaheuristic algorithms
13: **while** Stopping criteria are not reached **do** ▷ Iterate until the stopping criteria are fulfilled
14: **for** $n = 1 : N$ **do** ▷ Iterate over each individual in the current population
15: Update the individual using a global or local operator or both
16: **while** $Q_1(x) > 0$ **do** ▷ Check the ISPM size constraint
17: Repair the solution by removing a basic block from ISPM
18: **while** $Q_2(x) > 0$ **do** ▷ Check the DSPM size constraint
19: Repair the solution by removing a data variable from DSPM
20: Perform jump correction and evaluate the current individual
21: Update the next generation using the selection operator
22: Collect the Pareto optimal solutions obtained for p^{th} initial population
23: **Return:** Final Pareto optimal solution set (PF_F)

we repair the individuals by randomly placing basic blocks and global data objects back into the Flash memory until the constraints are satisfied (Lines 6-9). We perform jump correction and evaluate the initial population using WCET, energy, and schedulability analyzers (Lines 10-11). The schedulability analysis requires the WCET values of each task in the multi-task system. Therefore, we always perform schedulability analysis after performing WCET analysis.

After initialization, we initialize the hyperparameters required to run the metaheuristic algorithms (Line 12) and solve the MTMO problem by calling the metaheuristic algorithm (Lines 13-22). We iterate and update the population every generation until the stopping criteria are met (Line 13). For the MTMO problem, we consider two stopping criteria—the maximum number of generations and the maximum number of generations for which the population remains the same.

The metaheuristic algorithm uses local and global update operators to update individuals every generation (Line 15). As discussed in Section 4.4, each metaheuristic algorithm has its own local and global update operators. Again, we repair the updated individuals if the SPM size constraints are violated and evaluate them to collect their respective objective values (Lines 16-20). Based on the WCET, energy, and schedulability objective values and the concept of Pareto dominance (cf. Definition 4.1), the algorithm uses a selection operator to collect the top-scoring solutions and pass them to the next generation (Line 21).

After fulfilling the stopping criteria, the algorithm outputs the Pareto optimal solution set for the current initial population (Line 22). We collect all P Pareto fronts obtained after solving the MTMO problem for each initial population and generate the final Pareto optimal set (PF_T), the algorithm's output (Line 23).

Algorithm 7.1 considers all the basic blocks in the multi-task system. The number of basic blocks defines the dimension of the search space for the MTMO optimization problem. The second stopping criterion can be influenced by the search space size. Therefore, the smaller the search space size, the quicker we might reach the second stopping criterion, leading to fewer individuals to evaluate.

In the following two sections, we propose two techniques to reduce the size of the search space: PCA and ICA. By reducing the size of the search space, we aim to reduce the compilation time required to solve the MTMO optimization problem while relatively maintaining the quality of the solutions.

7.4 Path-based Constraint Approach (PCA)

While performing a multi-objective optimization, exploring the complete search space would be the most reliable way to find the Pareto optimal solutions. However, given the limited time under which we can perform any real-life optimization, we can consider objective-specific details to decrease the size of the problem.

WCET and energy objectives of a solution rely on the Worst-Case Execution Path (WCEP) (cf. Definition 3.4) and the Worst-Case Energy Consumption Path (WCECP) (cf. Definition 3.6) of a program, respectively. WCEP and WCECP are defined as the execution paths through a task's control flow graph that lead to their respective WCET and energy consumption values. While performing static SPM allocation, we know that the basic blocks on the WCEP and WCECP have the most impact on the WCET and energy objectives of the program. Therefore, instead of exploring the entire search space, we propose the Path-based Constraint Approach (PCA) that constrains the search space at each iteration using WCEP and WCECP information.

As the WCEP and WCECP of a task set can have different basic blocks, PCA considers the basic blocks on both paths. The WCEP and WCECP of a program

depend on the WCET and energy consumption of each basic block on the respective paths. For example, if we statically allocate a basic block that was previously on WCEP to ISPM, the WCET of that basic block will decrease due to its execution from ISPM, and that might lead to the removal of that basic block from the WCEP. Similar phenomena of path switching can also occur in the case of WCECP due to optimization. Therefore, the PCA technique proposed in this chapter needs to take into consideration such path switching during optimization.

Let $\mathbb{W}^{(g,n)}$ and $\mathbb{E}^{(g,n)}$ be the set of basic blocks on WCEP and WCECP of the n^{th} individual of the g^{th} generation, respectively, where $n = \overline{1,N}$, $g = \overline{1,maxGen}$, and N is the total number of individuals in the population. $maxGen$ is the maximum number of generations, which is one of the stopping criteria of the metaheuristic algorithm. Moreover, let $\mathbb{U}^{(g,n)} := \mathbb{W}^{(g,n)} \bigcup \mathbb{E}^{(g,n)}$ be the set of all basic blocks that are on both WCEP and WCECP. $x^{(g,n)}$ is the n^{th} individual solution vector in the population for the g^{th} generation. For each solution vector $x^{(g,n)}$, basic blocks on WCEP and WCECP can be different. Therefore, for the PCA technique, the multi-task multi-objective static SPM allocation optimization problem defined by Equation 7.1 is extended by adding the following constraint:

$$x_{t_i}^{(g+1,n)} = 0, \quad \text{if} \quad b_{t_i}^{(g,n)} \notin \mathbb{U}^{(g,n)} \quad \forall t, i \tag{7.8}$$

Algorithm 7.2 describes the Path-based Constraint Approach (PCA) proposed in this chapter. In Algorithm 7.1, at generation g, we evaluate all the individuals by performing WCET, energy, and schedulability analyses. The PCA technique is called after the evaluation stage while solving the MTMO problem.

The PCA technique takes an evaluated individual $x^{(g,n)}$ as an input and provides constraints for $x^{(g+1,n)}$ from the next generation (Lines 1-2). For PCA, we focus on the WCET and energy analysis results. As the results from the schedulability analyzer are at the task level, and as the dimension of the

Algorithm 7.2 Path-based Constraint Approach (PCA)

1: **Input:** Evaluated individual $x^{(g,n)}$
2: **Output:** Constraints for next individual $x^{(g+1,n)}$
3: Build the $\mathbb{W}^{(g,n)}$ and $\mathbb{E}^{(g,n)}$ sets by collecting basic blocks on the WCEP and WCECP of the evaluated individual $x^{(g,n)}$
4: Create $\mathbb{U}^{(g,n)}$ from the union of $\mathbb{W}^{(g,n)}$ and $\mathbb{E}^{(g,n)}$
5: **for** $t = 1 : T$ **do** ▷ Iterate over all the tasks
6: **for** $i = 1 : \tilde{d}_t$ **do** ▷ Iterate over all the basic blocks
7: **if** $b_{t_i}^{(g,n)} \notin \mathbb{U}^{(g,n)}$ **then**
8: $x_{t_i}^{(g+1,n)} = 0$

MTMO problem is defined by the basic blocks in the multi-task systems, we do not use schedulability analysis results to reduce the search space size.

The basic blocks on the WCEP and WCECP are used to build the sets of basic blocks $\mathbb{W}^{(g,n)}$ and $\mathbb{E}^{(g,n)}$ (Line 3). The union of $\mathbb{W}^{(g,n)}$ and $\mathbb{E}^{(g,n)}$ gives us $\mathbb{U}^{(g,n)}$ (Line 4). We iterate through all the tasks and their respective basic blocks. And if a basic block is not on WCEP or WCECP, then that decision variable of the individual vector for the next generation is constrained to 0, and the basic block is placed in the Flash, i.e., $x_{t_i}^{(g+1,n)} = 0$ (Lines 5-8). Therefore, the basic blocks not on WCEP or WCECP are placed in the Flash memory.

This constraint is enforced when the recombination and mutation operators for the SPEA algorithm and the local and global pollination operators for the FPA algorithm are used to update the individual to $(n+1)^{th}$ generation. Therefore, PCA considers the effect of the change in WCEP and WCECP for every individual during each iteration. The size and shape of the search space can change and be differently constrained throughout the optimization run.

7.5 Impact-based Constraint Approach (ICA)

The path-based constraint approach proposed in the previous section considered all basic blocks on the WCEP and WCECP to reduce the dimension of the search space. However, the MTMO optimization problem has an SPM size constraint, and SPMs are known for their small memory sizes. Consequently, we can assume that many basic blocks will not be assigned to SPM for large multi-task systems. Therefore, we propose the Impact-based Constraint Approach (ICA), which guides the optimization using the SPM size constraint, Worst-Case Execution Count (WCEC) (cf. Definition 3.5), and Worst-Case Energy Consumption Count (WCECC) (cf. Definition 3.7) to constrain the search space. WCEC and WCECC are the number of times each basic block is executed in scenarios that lead to WCET and worst-case energy consumption, respectively (cf. Section 3.2). As the schedulability objective operates at the task level, we do not consider it for search space reduction.

Algorithm 7.3 describes the initialization phase of the ICA technique. While initializing ICA, we calculate the impact of each basic block on the total WCET and energy consumption when SPM allocation is not performed. We consider an individual I_F where all the basic blocks are in the Flash and evaluate it by performing WCET and energy analyses (Line 1).

Let $\mathbb{W}^{I_F}$ be the set of ratios of the basic block's WCET to the total WCET for the individual I_F. Let $\mathbb{E}^{I_F}$ be the set of ratios of the basic block's energy consumption to the total energy consumption for I_F. After evaluating the individual I_F, we initialize the sets $\mathbb{W}^{I_F}$ and $\mathbb{E}^{I_F}$ and iterate over all the tasks and the respective basic blocks (Lines 2-4).

Let W^{I_F} and E^{I_F} be the total WCET and energy consumption of individual I_F. Let $W_{t_i}^{I_F}$ and $E_{t_i}^{I_F}$ be the WCET and energy consumption of the basic block $b_{t_i}^{I_F}$.

Algorithm 7.3 Initialization phase for the Impact-based Constraint Approach (ICA)

1: Create and evaluate an individual (I_F) with all the basic blocks in the Flash memory using WCET and energy analyzers
2: **Initialize:** $\mathbb{W}^{I_F}$ and $\mathbb{E}^{I_F}$
3: **for** $t = 1 : T$ **do** ▷ Iterate over all the tasks
4: **for** $i = 1 : \tilde{d}_t$ **do** ▷ Iterate over all the basic blocks
5: Collect WCET ($W^{I_F}_{t_i}$) and energy ($E^{I_F}_{t_i}$) values for basic block b_{t_i}
6: Calculate $\mathbf{W}^{I_F}_{t_i}$ and $\mathbf{E}^{I_F}_{t_i}$ metrics
7: $\mathbb{W}^{I_F} \longleftarrow \mathbf{W}^{I_F}_{t_i}$ and $\mathbb{E}^{I_F} \longleftarrow \mathbf{E}^{I_F}_{t_i}$

$b^{I_F}_{t_i}$ is the i^{th} basic block of the t^{th} task of the individual I_F. We collect WCET and energy values ($W^{I_F}_{t_i}$ and $E^{I_F}_{t_i}$) of each basic block when all basic blocks are placed in the Flash (Line 5). We calculate the ratio of the basic block's WCET and energy consumption to the total WCET and energy consumption of the program (Line 6) by using the following metrics:

$$\mathbf{W}^{I_F}_{t_i} := \frac{W^{I_F}_{t_i}}{W^{I_F}} \quad \text{and} \quad \mathbf{E}^{I_F}_{t_i} := \frac{E^{I_F}_{t_i}}{E^{I_F}} \tag{7.9}$$

After calculating $\mathbf{W}^{I_F}_{t_i}$ and $\mathbf{E}^{I_F}_{t_i}$ metrics for each basic block, they are added to $\mathbb{W}^{I_F}$ and $\mathbb{E}^{I_F}$ (Line 7). $\mathbf{W}^{I_F}_{t_i}$ and $\mathbf{E}^{I_F}_{t_i}$ metrics for each basic block in $\mathbb{W}^{I_F}$ and $\mathbb{E}^{I_F}$ are used to calculate the impact metrics while solving the MTMO problem using the ICA technique. To solve the MTMO problem using the ICA technique, we extend Algorithm 7.1 such that Algorithm 7.3 is called before initializing the MTMO problem in Algorithm 7.1. After initializing the ICA technique, we initialize the initial populations and run the metaheuristic algorithms using Algorithm 7.1.

Next, Algorithm 7.4 describes the proposed Impact-based Constraint Approach (ICA) used to reduce the size of the search space. The ICA algorithm takes an evaluated individual $x^{(g,n)}$ as an input and provides constraints for the individual $x^{(g+1,n)}$ from the next generation (Lines 1-2). For the ICA technique, we focus on the WCEC and WCECC values returned by the respective analyses. The ICA technique uses the following *impact* metric for each basic block to constrain the search space.

$$\mathbb{M}_{t_i} = \mathbf{W}^{I_F}_{t_i} * w^{(g,n)}_{t_i} + \mathbf{E}^{I_F}_{t_i} * e^{(g,n)}_{t_i} \tag{7.10}$$

The terms $w^{(g,n)}_{t_i}$ and $e^{(g,n)}_{t_i}$ in Equation 7.10 represent the WCEC and WCECC of the basic block $b^{(g,n)}_{t_i}$ of the individual n at generation g of the metaheuristic algorithm run, respectively.

Algorithm 7.4 Impact-based Constraint Approach (ICA)

1: **Input:** Evaluated individual $x^{(g,n)}$
2: **Output:** Constraints for next individual $x^{(g+1,n)}$
3: **Initialize:** $\mathcal{H}^{(g,n)}$ ▷ a set of the ordered pairs of basic blocks and their respective impact metrics
4: **for** $t = 1 : T$ **do** ▷ Iterate over all the tasks
5: **for** $i = 1 : \tilde{d}_t$ **do** ▷ Iterate over all the basic blocks
6: $\mathcal{H}^{(g,n)} \longleftarrow (BB_{t_i}^{(g,n)}, \mathbb{M}_{t_i})$
7: **Sort** $(\mathcal{H}^{(g,n)})$ in the descending order of $\mathbb{M}_{t_i}$ values
8: **Initialize:** $\mathcal{B}^{(g,n)}$ ▷ a list of unconstrained basic blocks
9: **for** $p = 1 : \tilde{d}$ **do** ▷ Iterate over $(\mathcal{H}^{(g,n)})$
10: **if** $(\mathcal{B}^{(g,n)} == \emptyset) \vee (\sum_{b=1}^{\eta} B_b <= S_{ISPM})$ **then**
11: $\mathcal{B}^{(g,n)} \longleftarrow \mathcal{H}_{(p,1)}^{(g,n)}$
12: **for** $t = 1 : T$ **do** ▷ Iterate over all the tasks
13: **for** $i = 1 : \tilde{d}_t$ **do** ▷ Iterate over all the basic blocks
14: **if** $BB_{t_i}^{(g,n)} \notin \mathcal{B}^{(g,n)}$ **then**
15: $x_{t_i}^{(g+1,n)} = 0$

While performing static SPM allocation, the basic blocks on the WCEP and WCECP have the most impact on the program's total WCET and energy consumption. However, basic blocks having higher WCEC and WCECC values do not indicate that the basic block will have higher WCET and energy consumption values. Therefore, the WCEC and WCECC terms ($w_{t_i}^{(g,n)}$ and $e_{t_i}^{(g,n)}$) in Equation 7.10 are multiplied by the WCET and energy impact terms ($\mathbf{W}_{t_i}^{I_F} \in \mathbb{W}^{I_F}$ and $\mathbf{E}_{t_i}^{I_F} \in \mathbb{E}^{I_F}$) from Equation 7.9 to calculate the basic block's impact. The bigger the value of $\mathbb{M}_{t_i}$, the higher the overall impact of the basic block $b_{t_i}^{(g,n)}$ on the total WCET and energy consumption of an individual.

We initialize an empty set $\mathcal{H}^{(g,n)}$ of the ordered pairs of the basic block and its *impact* metric (Line 3). We iterate through the multi-task system, i.e., $\tilde{d} = \sum_{t=1}^{T} \tilde{d}_t$ times, to fill the set $\mathcal{H}^{(g,n)}$ using each basic block $b_{t_i}^{(g,n)}$ and the respective impact metric $\mathbb{M}_{t_i}$ information (Lines 4-6). We sort this set in the descending order of the impact metric $\mathbb{M}_{t_i}$ values (Line 7).

We initialize an empty list $\mathcal{B}^{(g,n)}$, which will contain the list of basic blocks that will not be constrained by the ICA technique (Line 8). For selecting the basic blocks, we iterate through $\mathcal{H}^{(g,n)}$ (the set $\mathcal{H}^{(g,n)}$ is of size $\tilde{d}$ (Line 9)) and add basic blocks to $\mathcal{B}^{(g,n)}$ until the total size of the basic blocks selected, i.e., $\sum_{b=1}^{\eta} B_b$, is not greater than S_{ISPM}, where S_{ISPM} is the ISPM size (Lines 9-11). B_b is the size of the b^{th} basic block in $\mathcal{B}^{(g,n)}$. $\eta = |\mathcal{B}^{(g,n)}|$ is the current size of

$\mathcal{B}^{(g,n)}$, i.e., η represents the total number of basic blocks that are not constrained for the $(g+1)^{th}$ generation of the metaheuristic algorithm ($1 \leqslant \eta \leqslant d$).

Using this heuristic, we select the basic blocks that have the highest impact on the WCET and energy consumption of the code. We further constrain the search space by limiting the number of basic blocks selected by S_{ISPM}. If a basic block is not in $\mathcal{B}^{(g,n)}$, then that element of the individual for the next generation is constrained to 0, i.e., $x_{t_i}^{(g+1,n)} = 0$ (Lines 12-15). This constraint is enforced by SPEA's recombination and mutation operators and FPA's local and global pollination operators to update the individual to $(g+1)^{th}$ generation.

Therefore, if the basic block is not in $\mathcal{B}^{(g,n)}$, i.e., $b_{t_i}^{(g,n)} \notin \mathcal{B}^{(g,n)}$, then $x_{t_i}^{(g+1,n)}$ basic block of individual n will be placed in the Flash during the $(g+1)^{th}$ generation. Using this heuristic, we drastically constrain the search space of big multi-task problems and perform the MTMO optimization within a limited timeframe. As the ICA technique considers WCEC and WCECC information, it relies indirectly on the properties considered in the PCA technique. However, unlike PCA, the ICA technique does not ignore the basic blocks not on WCEP and WCECP, but they have a lower priority.

7.6 Evaluation

In this section, we perform evaluations to compare the MTMO approach with the PCA and ICA techniques in terms of compilation times and the quality of the obtained solutions. All approaches—MTMO, PCA, and ICA—are solved using two metaheuristic algorithms—FPA and SPEA. Moreover, we also compare these approaches with a baseline (BL), where the whole multi-task code is executed from the Flash memory.

Subsection 7.6.1 describes the evaluation setup used to solve the MTMO, PCA, and ICA optimization problems. Subsection 7.6.2 presents the compilation time results for the proposed MTMO, PCA, and ICA approaches. Subsection 7.6.3 discusses the Pareto optimal solutions found after solving MTMO, PCA, ICA, and BL evaluation runs. Subsection 7.6.4 compares the quality of MTMO, PCA, ICA, and BL solutions using the quality indicators described in Subsection 4.3.3.

7.6.1 Evaluation Setup

Similar to previous chapters, the evaluations are conducted using the WCC compiler framework for the ARM Cortex-M0 architecture. We use `aiT` v21.04i for WCET analysis, `EnergyAnalyzer` v21.04i for energy analysis, and an internal schedulability analyzer for schedulability analysis while solving the optimization problem (cf. Sections 3.2 and 3.3).

In Chapters 5 and 6, we performed evaluations using single-task benchmarks from the MRTC benchmark suite [GBEL10]. In this section, we evaluate multi-task systems. Within WCC, we can combine single-task benchmarks and

generate multi-task sets. WCC can automatically add suffixes to functions and global symbols to avoid duplicate definitions. Moreover, the main function within each single-task benchmark is prefixed and considered that task's *entry point* function. WCC is capable of generating a scheduler that can trigger individual tasks. However, as we are not dealing with evaluations on real hardware, it is not necessary to have a scheduler. We, therefore, can assume that one of the tasks in the multi-task system can act as a scheduler.

For these evaluations, we generated multi-task task sets that consist of 2, 4, 6, and 8 tasks each. For each task set size, 10 task sets were randomly generated. As mentioned before, we combine multiple single-tasking benchmarks into a multi-tasking task set for this purpose. We use the single-tasking benchmarks from the MRTC benchmark suite [GBEL10] with loop bound annotations from the TACLeBench project [FAH+16].

For schedulability analysis purposes, we need to define timing parameters like deadlines and periods for each multi-task task set. For these evaluations, we consider multi-task systems with periodic tasks (cf. Definition 2.6). Moreover, we use the EDF algorithm to perform schedulability analysis using the respective analyzer. To perform any meaningful evaluations, we want to generate multi-task systems that are not schedulable in their initial state, i.e., if the multi-task system is analyzed without applying any static SPM allocation-based optimization, the multi-task system is unschedulable.

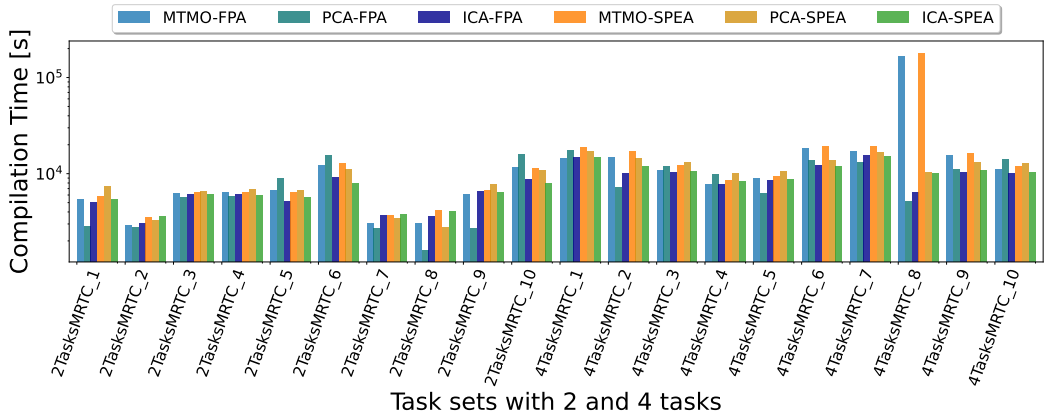
Within WCC, we can pass a user-defined overall system load parameter s_L that can be used to define tasks' properties like deadlines and periods. If s_L is above 1, the multi-task system is unschedulable by definition. s_L is used to automatically generate uniformly distributed partial loads (s_{L_t}) for each task (τ_t) of the multi-task system using the UUniFast algorithm [BB05]. The sum over all the partial loads s_{L_t} is equal to the user-defined load s_L.

Next, the period for each task (AP_t) is calculated using the WCET value W_t and the generated partial load s_{L_t} of each task, i.e., $AP_t = \frac{W_t}{s_{L_t}}$. It is important to note that schedulability analysis is performed after WCET analysis. Therefore, the WCET value W_t of every task is freely available for the schedulability analyzer's use. Similarly, the deadlines (D_t) are generated from a uniform random distribution between a range around the task periods. We can provide a user-defined lower (dl_l) and upper (dl_u) deadline range that is used to randomly decide a deadline D_t in the range $[dl_l AP_t, dl_u AP_t]$.

During these evaluations, we set s_L to 1.2, i.e., all the multi-task systems are initially unschedulable by design. Moreover, dl_l and dl_u are set to 0.8 and 1.2, respectively. The code is generated by WCC using the `-O2` compiler optimization flag. For reference sake, the names of the tasks in each task set are included in Table C.1 in Appendix C.

We set the SPM size to 60% relative to the code size of every task set to increase the pressure on the optimization. The initial population of any metaheuristic algorithm is defined randomly, and the results of the algorithm usually depend

Figure 7.1.: Compilation times required to perform MTMO, PCA, and ICA for task sets with 2 and 4 tasks, respectively



on the initial population. Therefore, we repeated the evaluations for both FPA and SPEA $P = 5$ times for different initial populations (cf. Algorithm 7.1). We combine all the individual Pareto fronts from the P optimization runs for P initial populations to find the final Pareto front (PF_F). We set a timeout value of 200 h for each multi-task set, and the compiler terminates the compilation if the optimizations have not finished successfully. Within this time limit, the task set is evaluated using both FPA and SPEA algorithms, and for each of the algorithms, the evaluation is repeated 5 times. In case a task set runs into a timeout, we collect the approximated Pareto optimal solutions from the final population of the algorithm as the final output.

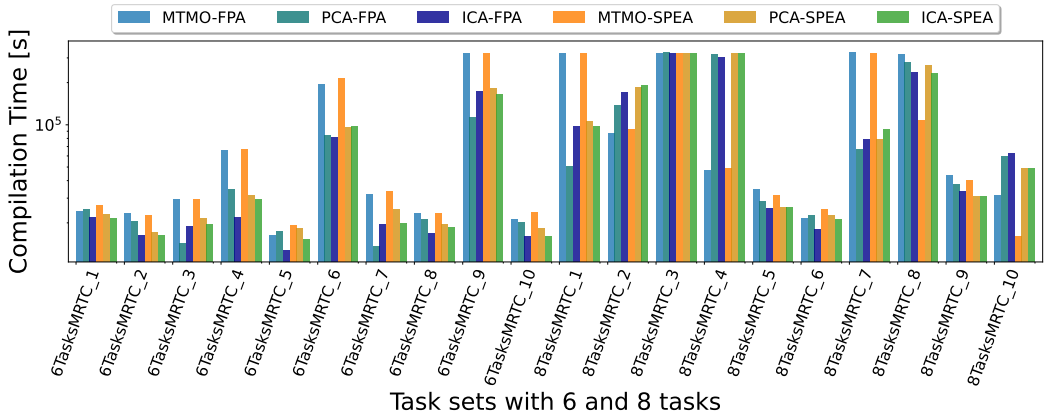
To solve the MTMO, PCA, and ICA optimizations using FPA and SPEA algorithms, we use the same hyperparameter values described in Tables 5.1 and 5.2 in Chapter 5. While performing evaluations for the PCA and ICA techniques, there is no need to set extra parameter values. The results from these evaluations are valid for the algorithm parameters described above.

7.6.2 Compilation Times

In this subsection, we compare the compilation times required by MTMO, PCA, and ICA evaluation runs. Figures 7.1 and 7.2 demonstrate the total compilation times to perform MTMO–FPA, MTMO–SPEA, PCA–FPA, PCA–SPEA, ICA–FPA, and ICA–SPEA optimizations in the form of a bar plot. The x-axis of the graphs represents the task sets, and the y-axis shows the compilation time required by the task sets to complete the optimization runs.

The compilation times shown here include all the iterations of each algorithm for $P = 5$ different initial populations mentioned above. The size of the task sets greatly influences the time required for the analyses of the objectives, i.e., WCET, energy consumption, and schedulability. The analysis of the objectives is one of the foremost factors determining the time required for the optimization.

Figure 7.2.: Compilation times required to perform MTMO, PCA, and ICA for task sets with 6 and 8 tasks, respectively



On average, MTMO is more time-consuming than PCA and ICA approaches. MTMO–FPA and MTMO–SPEA took, on average, 18.5 h and 17.37 h to finish the optimization runs, respectively. PCA–FPA and PCA–SPEA took, on average, 12.95 h and 14.33 h; ICA–FPA and ICA–SPEA, on average, took 13.28 h and 13.74 h to finish the optimizations, respectively. Therefore, we gained a 23.93% and 24.66% average reduction in compilation times using PCA and ICA over MTMO, respectively.

Even though there was an average reduction in compilation time using PCA and ICA, we observed that for some multi-task task sets, MTMO converged faster than PCA and ICA, especially for task sets with 8 tasks. For smaller task sets with 2, 4, and 6 tasks, on average, PCA and ICA were 47.65% and 49.38% faster than MTMO. However, we cannot derive any such definite conclusions between MTMO, PCA, and ICA's behavior for multi-task systems with 8 tasks.

Moreover, we observed that some task sets with 8 tasks ran into timeouts (cf. Figure 7.2). MTMO ran into the timeout for `8TasksMRTC_1`, `8TasksMRTC_3`, and `8TasksMRTC_7` multi-task task sets. PCA and ICA ran into the timeout for the `8TasksMRTC_3` task set. However, as mentioned before, when an optimization runs into a timeout, instead of crashing the optimization, we consider the current generation as the last generation, and we output the final Pareto from that generation.

We also compared the number of generations when the metaheuristic algorithms finished their optimizations. Due to the second stopping criterion, they can converge faster if there is no change in the population for 10 generations. On average, PCA and ICA converged in 5.09% and 0.23% less generations than MTMO, respectively.

As mentioned in Section 4.4, we maintain an archive of already evaluated individuals to avoid multiple evaluations of the same individuals. During these evaluations, MTMO, PCA, and ICA required, on average, 4.74%, 7.03%, and 1.9% fewer individual evaluations than the total number of individuals

due to already evaluated archived individuals. It implies that ICA encountered the same individuals less frequently than MTMO and PCA during its metaheuristic algorithm runs, i.e., we can say that ICA generates more varied individuals than PCA and MTMO.

Even though ICA conducted a larger number of evaluations compared to MTMO and PCA, ICA required less compilation time. The time for WCET, energy consumption, and schedulability analysis can vary based on the individuals under consideration. Moreover, schedulability analysis uses an ILP solver integrated within WCC to perform the respective analysis. Therefore, we cannot guarantee that each respective analysis can take the same amount of time. Consequently, unlike Chapter 6, we cannot conclude that PCA and ICA will always lead to a reduction in compilation times.

From overall observations, we can see that the second stopping criterion did not affect the SPEA algorithm too much, except for a few multi-task systems. The second stopping criterion affects FPA more; FPA converges before the maximum generation for most multi-task systems. Therefore, we can say that SPEA can maintain more variety in population generation-to-generation compared to FPA during these evaluations.

PCA and ICA reduce the search space size by constraining the decision vector. The total number of basic blocks in the task set determines the size of the search space. To calculate the reduction in the search space for PCA, we first calculated the total number of basic blocks not constrained by Equation 7.8, i.e., the total number of basic blocks on WCEP and WCECP for each individual in the population. As the number of basic blocks on WCEP and WCECP may vary, we calculated the average number of basic blocks on WCEP and WCECP over all the individuals and populations. Similarly, for ICA, we calculated the number of basic blocks that are not constrained according to Algorithm 7.4 and averaged over all the individuals and populations. For all task sets, the reduction in the search space achieved by using PCA and ICA is, on average, 23.97% and 52.83%, respectively.

`2TasksMRTC_2` has the smallest search space size of 28; PCA and ICA reduced this search space, on average, to 22 and 17, respectively. `8TasksMRTC_8` has the largest search space size of 612; PCA and ICA reduced this search space, on average, to 405 and 360, respectively. Even after reducing the search space using ICA, the smallest search space still leads to 2^{17} possible solutions.

The reduction in the search space can result in the algorithm hitting the second stopping criterion faster. However, the search space sizes are too large for the second stopping criterion to be effective every time. Therefore, it is difficult to find a direct correlation between the reduction in search space size, the second stopping criterion, and the reduction in the compilation time of the task sets. However, PCA and ICA were, on average, still faster than MTMO during these evaluations.

The second stopping criterion considered in this chapter might be too strict, i.e., if one individual is updated within a population, the condition for the

second stopping criterion is reset. In the future, we could use more relaxed stopping criteria, allowing us to terminate the metaheuristic algorithms more effectively. We could use the following two potential stopping criteria in the future [ES15]:

- If the population diversity drops under a given threshold, the metaheuristic algorithm is terminated.
- If the fitness improvement remains under a given threshold for a given period or a given number of generations, the metaheuristic algorithm is terminated.

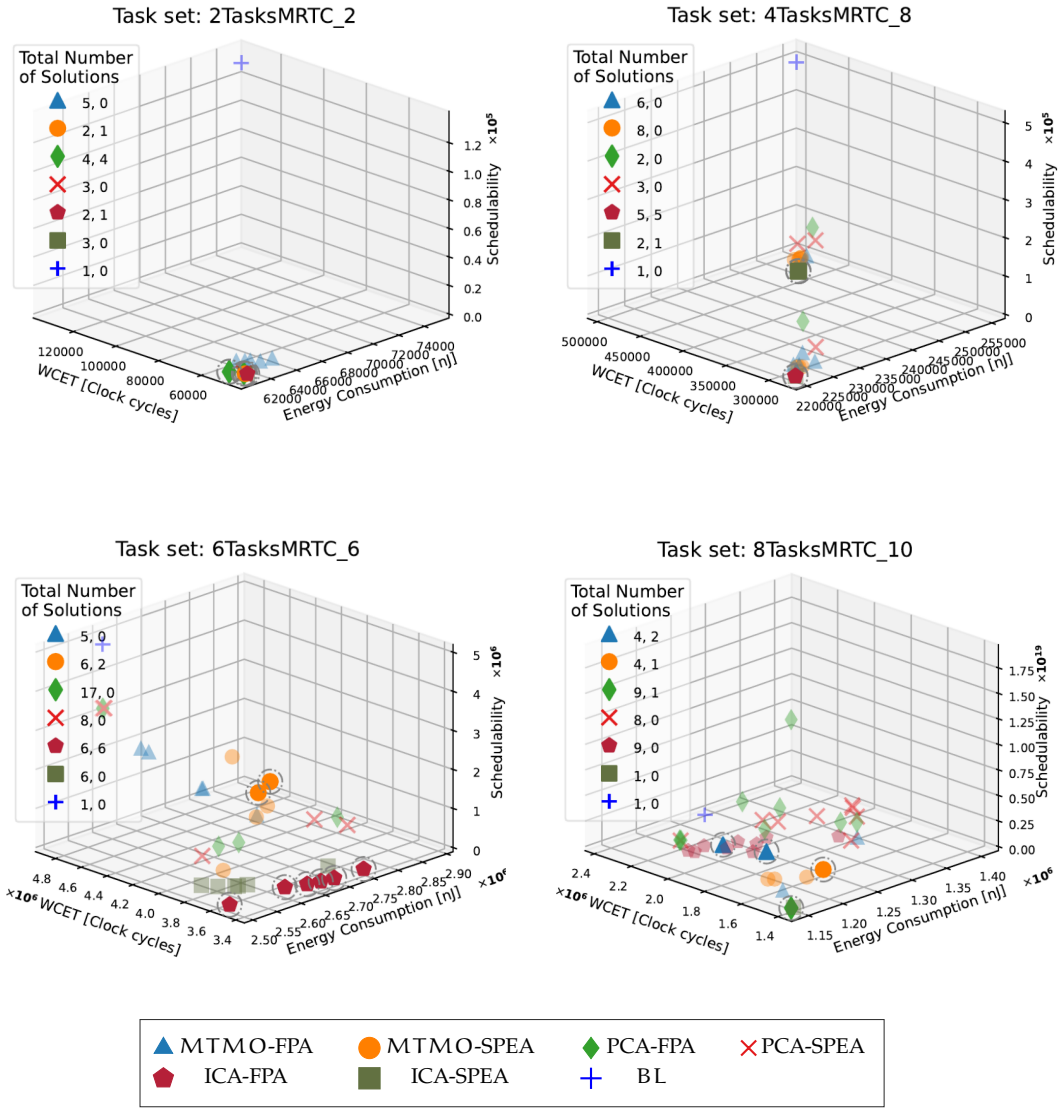
7.6.3 Pareto Fronts

In this subsection, we compare all the solutions obtained after performing MTMO, PCA, and ICA optimization runs. We also perform a baseline BL evaluation run and compare it with the respective optimization runs. Figure 7.3 shows the solutions found by all three approaches—MTMO, PCA, and ICA—and the respective metaheuristic algorithms—SPEA and FPA—in the form of Pareto fronts. The figure also shows the solution from the BL evaluation run, where all the tasks in the task sets are placed in the Flash memory. For the sake of brevity, this subsection presents the solutions for four task sets, one task set each with 2, 4, 6, and 8 tasks. The figures for all the remaining task sets are in Appendix C.

As mentioned in Section 4.3, obtaining the true Pareto front for a complex multi-objective optimization problem like MTMO is ambitious. Therefore, under a realistic assumption that the true Pareto front is unknown, we obtain an approximated true Pareto front (PF_T) from the union of all the final Pareto optimal solutions (PF_F) obtained from all the approaches. The multi-objective optimization problem we are dealing with has three objectives—WCET, energy, and schedulability. Therefore, the obtained solutions lie in a 3-dimensional objective space, represented by a 3D scatter plot.

In Figure 7.3, we represent the total WCET and the total energy values of the task sets on the x- and y-axis, respectively. The schedulability objective of the task sets is shown on the z-axis of the figure. In the figure, we can deduce that the solutions in the front-right corner of the solution space should be the intended Pareto solutions for our minimization problem. The title of each plot is the name of the task sets. The legend at the bottom of the figure presents the algorithm and the approach to which a particular solution belongs. In the legend of each sub-figure, we show the total number of solutions returned by each approach on their respective PF_Fs. In the same legend, we show the number of solutions contributed to the final PF_T by the respective approach. The solutions on PF_T are represented by darker colors in the figure, while the solutions on PF_F are depicted by the fainter version of

Figure 7.3.: Pareto fronts obtained using MTMO, PCA, ICA, and BL



those colors. Furthermore, for clarity, the solutions on PF_T are highlighted by dotted gray circles.

For the task set `2TasksMRTC_2` (cf. top-left sub-figure of Figure 7.3), we see that only MTMO–SPEA, PCA–FPA, and ICA–FPA contributed solutions to the final Pareto front PF_T. MTMO–SPEA and ICA–FPA contributed one solution to PF_T each; all the solutions from PCA–FPA's PF_F are on the final PF_T. Moreover, all the solutions on PF_T are completely schedulable, with zero schedulability objective value.

For the task set `4TasksMRTC_8` (cf. top-right sub-figure of Figure 7.3), only the ICA approach contributed solutions to the final PF_T. ICA–SPEA contributed one solution to PF_T, whereas all the solutions from ICA–FPA's PF_F are on the final PF_T. The ICA–SPEA solution is schedulable if one task is removed from

Table 7.1.: Comparison between the average number of solutions (in %) on the final PF_T from the respective PF_F of each optimization approach

MTMO		PCA		ICA		BL
FPA	SPEA	FPA	SPEA	FPA	SPEA	
18.56	29.29	2.45	1.63	28.29	1.41	0

Table 7.2.: Average reduction in WCET, energy consumption, and schedulability (in %) obtained using MTMO, PCA, and ICA over BL

Objectives	MTMO		PCA		ICA	
	FPA	SPEA	FPA	SPEA	FPA	SPEA
WCET	30.37	32.77	16.03	18.83	29.24	27.59
Energy consumption	4.55	5.45	1.2	1.22	3.22	2.11
Schedulability	79.48	78.34	59.69	69.3	78.13	80.22

the task set. However, two out of five solutions contributed by ICA–FPA are completely schedulable.

For the task set `6TasksMRTC_6` (cf. bottom-left sub-figure of Figure 7.3), MTMO–SPEA contributed two solutions PF_T out of six solutions from its PF_F. ICA–FPA contributed all six solutions from PF_F to the final PF_T. Moreover, the two MTMO–SPEA solutions on PF_T are partially schedulable. However, all six ICA–FPA solutions are completely schedulable.

Lastly, for the task set `8TasksMRTC_10` (cf. bottom-right sub-figure of Figure 7.3), MTMO–FPA provided two solutions to PF_T, and MTMO–SPEA and PCA–FPA provided one solution each to PF_T. Moreover, all the solutions on the final PF_T are completely schedulable. All Pareto fronts of the remaining task sets are presented in Figures C.1, C.2, C.3, C.4, and C.5 in Appendix C.

From overall evaluations, we calculated the total number of solutions provided by each evaluation run on the final PF_T, and we averaged it over all the task sets. Table 7.1 provides the average number of solutions (in %) on PF_T out of the total number of solutions on their respective approach's PF_F. From the table, we can see that MTMO–SPEA provides the highest percentage of solutions on PF_T from its PF_F. However, ICA–FPA provided slightly more solutions on the final PF_T. Moreover, ICA–FPA provided better solutions for most task sets, as evident from the next subsection.

We also compared the reduction in WCET, energy consumption, and schedulability objectives achieved using MTMO, PCA, and ICA over the baseline BL evaluation. Table 7.2 reports the percentage reduction in the objective values obtained using the proposed approaches over BL. In Table 7.2, percentage reductions are computed as the reduction of average objective values across all benchmarks, relative to the BL evaluations. As mentioned before, the task sets are designed to be unschedulable in their initial state. Consequently, the largest reduction is observed in the schedulability objective relative to

Table 7.3.: Total number of benchmarks with best (or non-dominated) quality indicators for each optimization approach

Quality Indicators	MTMO		PCA		ICA		BL
	FPA	SPEA	FPA	SPEA	FPA	SPEA	
C	8 (3)	9 (2)	2 (0)	2 (1)	15 (2)	0 (0)	0 (0)
NR	6 (6)	11 (3)	2 (0)	0 (0)	14 (3)	0 (0)	0 (0)
RNS	8 (3)	9 (2)	2 (0)	2 (1)	15 (2)	0 (0)	0 (0)
HV	10 (1)	8 (1)	1 (0)	2 (0)	15 (1)	2 (1)	0 (0)

BL. However, since the reduction in the schedulability objective is not 100%, this indicates that the resulting Pareto front solutions include a mixture of completely and partially scheduled solutions. From Table 7.2, we can also observe that the reduction in WCET is higher than that in energy consumption for all approaches. Moreover, MTMO and ICA achieved larger reductions in the respective objectives compared to the PCA approach.

7.6.4 Quality Indicators

As obtaining the true Pareto front is ambitious, for the sake of brevity, we consider the final Pareto front PF_T discussed in the previous subsection as our reference true Pareto front to calculate the quality indicators. To evaluate and compare the quality of solutions for MTMO, PCA, ICA, and the BL evaluation, we use four different quality indicators: Coverage (C), Non-dominance Ratio (NR), Ratio of Non-dominated Solutions (RNS), and Hypervolume (HV) (cf. Subsection 4.3.3).

Table 7.3 summarizes the total number of task sets with best or non-dominated quality indicators for MTMO, PCA, ICA, and BL. Table 7.3 is structured similarly to Tables 5.5 and 6.2. In Table 7.3, the numbers shown outside the bracket represent the total number of task sets with the best quality indicators, and the numbers shown inside the bracket represent the number of task sets with non-dominated Pareto optimal solutions compared to the other approaches. For reference sake, Tables C.2 and C.3 in Appendix C present the quality indicator values for all the evaluated task sets from these evaluations.

From Table 7.3, we can see that ICA–FPA performed better for more task sets than all other approaches. MTMO–FPA and MTMO–SPEA performed almost equally well, MTMO–SPEA with a slightly better performance. However, PCA and ICA–SPEA did not perform as well as ICA–FPA and MTMO, providing the least number of task sets with the best or non-dominated quality indicators. Moreover, we can see that all the approaches provided better quality solutions than BL for all task sets.

From overall evaluations, we observed that PCA provided the largest number of solutions on its respective PF_F and the least number of solutions on the final PF_T. Therefore, we can say that PCA had difficulty converging to better quality solutions compared to MTMO and ICA. Moreover, ICA–SPEA provided good

quality solutions overall. However, ICA–FPA and MTMO almost always outperformed ICA–SPEA.

From an overall comparison, we can see that ICA–FPA provided better quality solutions for most task sets faster than the other approaches. Although it is difficult to know beforehand which approach will deliver the better Pareto optimal solutions due to the non-deterministic nature of algorithms, we could find better results by constraining the search space and focusing the optimization direction on the basic blocks that highly impact the objectives.

7.7 Conclusion

In this chapter, we formulated a 3-dimensional SPM allocation-based MTMO problem that minimizes WCET, energy consumption, and schedulability objectives simultaneously. We solved the MTMO problem using two metaheuristic algorithms—FPA and SPEA. As the compilation times required for the optimization can increase with the problem size, we proposed two new approaches, PCA and ICA, to cope with the MTMO problem size. All these approaches were able to find the trade-offs between schedulability, WCET, and energy consumption. Moreover, we compared the results obtained using FPA and SPEA for all three approaches.

From evaluations, we achieved, on average, a 23.93% and 24.66% reduction in compilation times using PCA and ICA compared to MTMO, respectively. The MTMO approach provided good quality solutions for some task sets but was time-consuming. Using PCA and ICA, we achieved, on average, 23.97% and 52.83% reduction in the search space dimension compared to MTMO. However, as the MTMO search space was extremely large for almost all the task sets, the second stopping criterion was not as effective every time.

Although PCA was faster than MTMO, it returned comparatively poorer-quality solutions. However, ICA–FPA found good quality solutions with better quality indicators for the largest number of task sets compared to the other approaches. Conclusively, we can say that ICA-FPA was the best approach to getting quality solutions in less time during these evaluations. In this chapter, we showed that clever integration of the WCET- and energy-related information within the optimization leads to a compilation time reduction and helps find better-quality solutions.

PCA and ICA reduce the MTMO problem's search space using the WCET and energy objective's underlying properties. In this chapter, PCA and ICA focus only on the search space dimensions defined by the basic blocks. PCA and ICA do not consider the global data objects for further reduction of the search space. In the future, we could expand PCA and ICA to identify unnecessary global data objects and further reduce the MTMO problem's search space. Similar to Chapter 6, a separate analysis that identifies the number of accesses to the global data objects from the Flash and SPM could be used to estimate the cost incurred due to them. Moreover, we could use a

metric similar to the *impact* metric to select the global data objects with the most impact, and the rest could be constrained.

The approximation model proposed in Chapter 6 is more effective at reducing the compilation times required for solving the proposed multi-objective optimization. However, the PCA and ICA approaches proposed in this chapter can effectively reduce the search space of the optimization problem. Therefore, we can easily combine the PCA or ICA approach with the approximation model to get the benefits from both approaches.

PCA or ICA can help reduce the search space of the multi-objective optimization problem by constraining the search space, and the approximation model can be used to analyze each individual while running the metaheuristic algorithm. However, PCA and ICA inherently consider that the WCEP and WCECP of a program can vary during optimization, and the respective WCEP and WCECP information is derived from the WCET and energy analyses of the individual using respective analyzers. As the approximation models proposed in Chapter 6 do not have the notions of WCEP and WCECP, PCA and ICA might need to assume constant WCEP and WCECP to use the approximation model. Therefore, PCA or ICA can constrain the solution space once before starting the metaheuristic algorithm run. Furthermore, while running the metaheuristic algorithm, we can use the approximation model to calculate the objective values instead of the respective analyzers.

In Chapters 5, 6, and 7, we mainly focus on the multi-objective optimizations for compiler-level static SPM allocation. We also proposed various techniques to deal with the compilation time bottleneck that we encountered while solving the multi-objective optimization. However, static SPM allocation has an SPM size constraint, which limits the number of basic blocks that can be allocated to the SPM statically and hinders the possible optimization potential. In the next chapter, we propose a compiler-level multi-objective dynamic SPM allocation optimization that helps us to deal with the SPM size constraint. Instead of statically allocating basic blocks once during the startup, the approach proposed in the next chapter enables the dynamic allocation of objects from the Flash memory to SPM during runtime.

Part III

Dynamic Allocation

8

Multi-Objective Dynamic SPM Allocation

In Chapters 5, 6, and 7, we focused on compiler-level static SPM allocation. As SPMs are fast and energy-efficient local memories, the multi-objective static SPM allocation-based optimizations we explored showed great optimization potential. However, SPM's small size gravely constrains the static optimization problem.

Contrary to SPMs, as caches can autonomously adapt and change the contents within them during execution, they are not constrained by their sizes. However, caches are notoriously unpredictable and pose a reliable analysis challenge for real-time systems at compile time. On the other hand, SPMs are predictable but cannot adapt to changing execution scenarios during runtime, like caches. Therefore, to circumvent the SPM size constraint, to adapt flexibly to changing execution scenarios, to support the re-use of SPM memory regions, and to do so predictably, we propose a compiler-level dynamic SPM allocation model in this chapter.

Dynamic SPM allocation is a compiler optimization used to allocate objects dynamically from one memory to another during runtime. Using dynamic SPM allocation, we can change the contents of SPM during runtime and predictably analyze the runtime behavior, unlike caches. However, we encounter the following issues while performing compiler-level dynamic SPM allocation:

- Code objects can have overlapping executions. For example, the basic blocks of a loop in a nested loop have overlapping execution with basic blocks in other loops of that nested loop.
- Two code objects with overlapping executions cannot occupy the same address space within SPM during runtime.
- The static WCET and energy analysis tools cannot accommodate dynamic copying of code at compile-time analysis.

In this chapter, we resolve the issues mentioned above by proposing the following, which enables us to compile and analyze dynamically allocatable code at compile time.

- We combine a liveness analysis with the proposed dynamic SPM allocation model to get live ranges of dynamic allocation objects.

- We proposed an address assignment algorithm that uses a combined first- and best-fit heuristic to find appropriate addresses within SPM for the dynamic allocation objects to generate valid dynamically allocatable code.
- We propose a strategy to perform WCET and energy analyses of such dynamically allocated programs statically at compile-time, enabling us to perform dynamic SPM allocation-based multi-objective optimization during compilation.

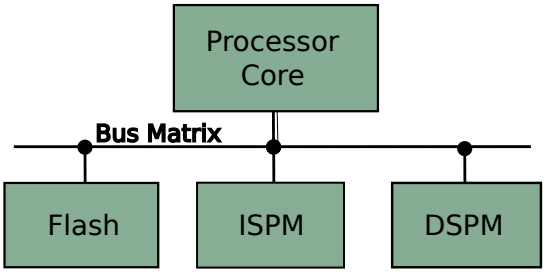
For static allocation techniques, the content in SPM is written once during startup and never changed during runtime. On the contrary, for dynamic allocation, even though we decide which object is placed within the SPM at compile time, the content in the SPM is dynamically overwritten during runtime. However, this process presents some challenges. We need to perform code transformations to jump from one memory to another, which incurs extra WCET and energy consumption overheads. Functions and loops tend to be larger in size and are executed more frequently compared to individual basic blocks. As a result, executing these larger constructs from SPM can compensate for the overhead due to their dynamic allocation. Therefore, in this thesis, we consider functions and loops as dynamic allocation candidates. Moreover, we perform static allocation of global data variables. However, local data variables are beyond the scope of this thesis.

Dynamic allocation is traditionally an important task for operating systems. However, the execution times of operating systems-based memory allocation techniques are difficult to predict and to give guarantees. Therefore, approaches like free-list management and deallocation are not feasible in the realm of real-time systems. However, compiler-level dynamic allocation can be exploited to compile WCET- and energy-efficient dynamically allocatable code for real-time embedded systems.

To the best of our knowledge, only dynamic allocation-based single-objective optimizations have been considered until now to minimize either WCET or energy consumption. Therefore, in this chapter, we propose a multi-objective optimization that uses the proposed dynamic SPM allocation model and simultaneously optimizes the program's WCET and energy consumption.

Dynamic SPM allocation, in itself, is an NP-hard problem; i.e., we can draw parallels between the dynamic SPM allocation problem and a bin packing problem, which is an NP-hard problem. To solve such a dynamic allocation problem, we need to find the optimal assignment of dynamic allocation objects that minimizes the program's WCET and energy consumption. The simultaneous consideration of multiple objectives and the overheads occurring due to that adds extra expenses to the already complex problem. To solve such multi-objective optimization, we implemented the proposed dynamic SPM allocation-based multi-objective optimization within the WCC compiler framework, and we solved it using two metaheuristic algorithms—FPA and

Figure 8.1.: An exemplary architecture for dynamic SPM allocation



SPEA. For brevity's sake, we hereafter refer to the multi-objective dynamic SPM allocation optimization run as MO_D in this chapter.

Figure 8.1 shows an exemplary architecture that is considered in this chapter for dynamic SPM allocation. This underlying architecture consists of a Flash memory, an instruction SPM (ISPM), and a data SPM (DSPM). Unless explicitly specified, an SPM always refers to the ISPM in this chapter. During evaluations, we generate code, perform analysis, and perform MO_D optimization for ARM Cortex-M0. Figure 8.1 exactly represents the STM32F0xx implementation of the ARM Cortex-M0 architecture [STM22a].

In the past, compiler-based dynamic memory allocation techniques have been investigated for architectures with a Flash and an SPM. Even though we use the underlying architecture from Figure 8.1 during optimization, the approach proposed in this chapter can easily be expanded to support DRAM, SRAM, or both if their timing and energy characterizations are available.

The dynamic SPM allocation model and the MO_D problem proposed in this chapter have architecture-independent and architecture-dependent parts. The solutions, objectives, liveness analysis, minimization problem formulation, and metaheuristic algorithms are architecture-independent. However, the code compiled by WCC, the code transformations, and objective analyses are architecture-dependent. Although the address assignment algorithm proposed in this chapter takes inputs as the underlying memory hierarchy, the assembly-level code, and the memory addresses of the dynamically allocatable objects, the address assignment algorithm can be used for other architectures.

The multi-objective dynamic SPM allocation proposed in this chapter was presented at the International Workshop on Worst-Case Execution Time Analysis (WCET) in 2023 [JF23b]. Moreover, some multi-objective dynamic SPM allocation-related details are also published in Springer's International Journal of Time-Critical Computing Systems (Real-Time Systems) in 2025 [JF25]. The key contributions of this chapter are:

- We formulate a compiler-level dynamic SPM allocation model that enables the allocation of objects from the Flash memory to SPM during runtime and supports both WCET and energy analyses of dynamically allocatable code.

- We propose a dynamic SPM allocation-based multi-objective optimization (MO_D) that aims to minimize the program's WCET and energy consumption simultaneously.
- We solve the MO_D problem using two metaheuristic algorithms—FPA and SPEA.

During evaluations, we demonstrate the utility of the proposed MO_D optimization by comparing its evaluation results with the static SPM allocation-based multi-objective optimization (MO_S) from Chapter 5 and a state-of-the-art ILP-based dynamic SPM allocation-based single-objective optimization [VM07], henceforth referred to as SO_D. We also compare MO_D with a baseline (BL) evaluation, where the whole program is executed from the Flash memory. We compare MO_D with MO_S and SO_D using real-world benchmark suites offered by the `Embedded Microprocessor Benchmark Consortium (EEMBC)` [PCLGO09]. The key results from the evaluations are:

- As dynamic SPM allocation introduces significant WCET and energy overheads, the quality of solutions from MO_D is only slightly worse than the ones from MO_S.
- MO_D and MO_S clearly outperform the state-of-the-art SO_D and the BL evaluation.

This chapter is organized as follows: Section 8.1 provides an overview of the related background work. Section 8.2 proposed the problem formulation for the compiler-level dynamic SPM allocation model. Section 8.3 presents liveness analysis, address assignment algorithm, code generation, and code analysis for the proposed dynamic SPM allocation. Section 8.4 presents the metaheuristic algorithm-based MO_D optimization. Section 8.5 presents the evaluation results, and Section 8.6 concludes the chapter.

8.1 Related Work

Dynamic SPM allocation facilitates copying objects on and off memories during runtime and exploits the memory subsystem to its fullest potential. Therefore, dynamic allocation introduces added complexity compared to static allocation in determining which object can be copied to which memory location within the memory hierarchy and at what point during execution.

Anand et al. [AB15] presented a heuristics-based instruction cache locking mechanism for improving the average-case runtime of an embedded system by updating the addresses locked in the cache dynamically during the execution of a program. Puaut et al. [PA06] explored the use of local dynamic locking of instruction caches to improve the worst-case performance of a system. They proposed algorithms to partition the machine code into regions and to

determine a locked state of the instruction cache for each such region. Ding et al. [DLM14] proposed a loop-based dynamic instruction cache locking technique for minimizing WCET. They used an ILP-based global optimization method to perform dynamic cache locking. These approaches use cache locking mechanisms to improve real-time predictability, making caches nearly functionally equivalent to SPMs. Instead of locking caches and disregarding the benefits of cache at the cost of predictability, we use SPMs in this chapter to predictably place code in SPMs during runtime.

Pellizzoni et al. [PBB+11] proposed a PRedictable Execution Model (PREM) that divides the execution of a program into two predictable phases to ensure timing predictability. In the memory phase, the required data is pre-loaded into local memory or caches. In the execution phase, the program runs using the data pre-loaded in the memory phase. Mancuso et al. [MDC14] introduced the light-PREM that automated the refactoring process needed to convert the legacy software code to PREM-compliant code. However, unlike PREMs, dynamic SPM allocation allows changing and rewriting the content within SPM during runtime.

In the past, dynamic SPM allocation-based approaches were proposed in the context of single-objective optimizations [KRI+01, SP17, PFV+07, FMA+04, QCM+17]. Kandemir et al. [KRI+01] presented a compiler-level dynamic SPM management framework that uses loop and data transformations to utilize the SPM as efficiently as possible. Soliman et al. [SP17] proposed a compiler-directed prefetching scheme for SPM, which efficiently optimizes the dynamic allocation of program data to minimize the program's WCET. Deverge et al. [DP07] proposed a dynamic SPM allocation algorithm for static and stack data to minimize WCET. Kim et al. [KBS17] proposed dynamic instruction allocation at the function level for minimizing WCET using direct memory access transfers.

Verma et al. [VM07] proposed a scratchpad overlay approach for data and instruction allocation that minimizes the energy consumption of the program (SO_D). They proposed a two-step approach, where the first step assigns objects to SPM and determines optimal locations to insert code for copying objects during runtime. The second step computes addresses for objects assigned to the SPM. The ILP formulation used for the address assignment problem required considerable time to compute the solutions for large SPM sizes and larger benchmarks. Therefore, they proposed a near-optimal approach, replacing the ILP formulation with a first-fit heuristic. During evaluations, we compared the proposed MO_D optimization with Verma's SO_D approach. Liu et al. [LZ15] built upon Verma's scratchpad overlay model and considered a multi-level SPM architecture for multi-core processors to minimize WCET.

Francesco et al. [FMA+04] exploited hardware support to alleviate the transfer costs to and from SPM to minimize energy consumption and provided a high-level programming interface for runtime SPM management. Qiu et al. [QCM+17] presented an approach to reduce the energy consumption

for a system with a hybrid SPM architecture by deriving a suitable data allocation using dynamic programming—hybrid SPM incorporates SRAM, magnetic RAM, and zero-capacitor RAM for overall performance gain. Wang et al. [WLL18] proposed a dynamic data allocation strategy for a hybrid SPM with SRAM and NVM to solve the task scheduling problem. Pyka et al. [PFV+07] proposed dynamic SPM allocation for data objects directly integrated into a real-time operating system to reduce energy consumption.

These approaches consider the dynamic SPM allocation of either code or data and focus on only single-objective optimization. However, none of these approaches focus on dynamic allocation-based multi-objective optimization to exploit the memory subsystems and optimize multiple objectives simultaneously. Therefore, in this chapter, we perform multi-objective dynamic SPM allocation that simultaneously minimizes the WCET and energy consumption of the program.

8.2 Problem Formulation

In this section, we provide definitions related to dynamic SPM allocation and formulate the multi-objective dynamic SPM allocation optimization problem. Dynamic allocation is a low-level compiler optimization that enables the dynamic movement of objects from one memory to another during runtime.

This section is divided into the following subsections: Subsection 8.2.1 defines the memory objects that are dynamically allocated during runtime from one memory to another. Subsection 8.2.2 describes the decision variable vector associated with dynamic SPM allocation. Subsection 8.2.3 defines the address objects needed to solve the address assignment algorithm. Subsection 8.2.4 provides a mathematical formulation for the dynamic SPM allocation-based multi-objective optimization problem that is solved in this chapter.

8.2.1 Memory Objects

In this thesis, we consider so-called memory objects as dynamic allocation candidates. The memory object is the finest granularity of code fragments considered for dynamic allocation. Dynamic allocation of memory objects requires extra code and incurs WCET and energy overheads, i.e., to dynamically allocate memory objects during runtime, we need to add extra code to actually copy memory objects from the Flash memory to SPM during runtime. Moreover, we need to add the extra jump correction code that enables jumping of code between memories. Thus, memory objects should ideally be big and executed multiple times in SPM to compensate for the additional overheads. Therefore, we consider functions and loops as memory object candidates that we dynamically allocate from the Flash to ISPM during runtime. We define the memory objects as follows:

Definition 8.1 (Memory Object)
A memory object is defined as a tuple $\mathcal{M}_{(p,m)} = (\mathcal{B}_{(p,m)}, \zeta)$*, where* $\mathcal{B}_{(p,m)}$ *is a set of basic blocks associated with* $\mathcal{M}_{(p,m)}$*—the* m^{th} *memory object in the* p^{th} *function—and* ζ *is a predicate that indicates whether* $\mathcal{B}_{(p,m)}$ *constitutes a function or a loop.*

The set of basic blocks associated with either a function or a loop defines the respective memory object. Finally, let $\texttt{MO} = \{\mathcal{M}_{(1,1)}, \ldots, \mathcal{M}_{(1,M_1)}, \ldots, \mathcal{M}_{(P,M_P)}\}$ be the set of memory objects, where $\mathcal{M}_{(p,m)}$ is the m^{th} memory object in the p^{th} function. P represents the total number of functions, and M_p is the total number of memory objects in the p^{th} function.

The global data variables (hereafter referred to as global data objects) are live for the entire program execution. Consequently, we do not see the need for the dynamic allocation of global data objects during runtime. Therefore, for brevity's sake, we consider them static allocation candidates and allocate them statically to DSPM.

Let $\mathcal{G}_g$ be the g^{th} global data object within the program. Therefore, in this chapter, the memory allocation model considers the dynamic allocation of memory objects ($\mathcal{M}_{(p,m)}$) from the Flash to ISPM during runtime and the static allocation of global data objects ($\mathcal{G}_g$) from the Flash to DSPM using startup code. We don't consider local data object allocation since they are usually stored on the stack. Dynamic allocation of local data would come with heavy stack frame reorganization, which is beyond the scope of this thesis.

8.2.2 Decision Variable Vector

The dynamic allocation problem is formulated as a binary decision problem, where we need to decide which memory object of the program should be allocated dynamically to SPM during runtime. The decision to move a particular memory object to SPM depends on the underlying minimization objectives—we consider WCET and the energy consumption of a program as the minimization objectives.

Let $x = (x_{(1,1)}, .., x_{(1,M_1)}, .., x_{(P,M_P)}, x_{(\tilde{d}+1)}, .., x_d) \in X$ be a d-dimensional binary decision variable vector, where $X \subset \{0,1\}^d$ is the decision or search space of the multi-objective dynamic SPM allocation problem. P represents the total number of functions in the program, and M_p is the total number of memory objects in the p^{th} function. Lastly, d is the total dimension of the search space, and $\tilde{d}$ is the total number of memory objects (functions and loops) within the program, i.e., $\tilde{d} = \sum_{p=1}^{P} M_p$.

The subvector $x_{(1,1):(P,M_P)} \subset \{0,1\}^{\tilde{d}}$ represents the dynamic allocation decision vector for memory objects, and $x_{(p,m)} \in \{0,1\}$ decides if a memory object will be allocated dynamically to ISPM during runtime or not. I.e., if $x_{(p,m)} = 0$, the memory object $\mathcal{M}_{(p,m)}$ will stay in the Flash memory, and if $x_{(p,m)} = 1$, $\mathcal{M}_{(p,m)}$ is allocated to ISPM during runtime. Depending on this decision

subvector, the memory objects are transferred dynamically to ISPM during runtime.

$$x_{(p,m)} = \begin{cases} 1, & \text{if } \mathcal{M}_{(p,m)} \text{ is dynamically allocated to ISPM} \\ 0, & \text{otherwise} \end{cases} \tag{8.1}$$

For each function f_p, $x_{(p,1)}$ is the decision variable for the function itself, and the remaining decision variables, i.e., $x_{(p,2):(p,M_p)}$, are decisions for the loops within the p^{th} function.

$$\forall g = \overline{(\tilde{d}+1), d} \quad x_g = \begin{cases} 1, & \text{if } \mathcal{G}_g \text{ is statically allocated to DSPM} \\ 0, & \text{otherwise} \end{cases} \tag{8.2}$$

The subvector $x_{(\tilde{d}+1):d} \subset \{0,1\}^{(d-\tilde{d})}$ represents the static allocation decision vector for global data variables, where $(d-\tilde{d})$ is the total number of global data variables. $x_g \in \{0,1\}\, \forall g = \overline{(\tilde{d}+1), d}$ decides if a global data object will be statically allocated to DSPM at compile-time or not. I.e., if $x_g = 0$, the global data object $\mathcal{G}_g$ will stay in the Flash memory, and if $x_g = 1$, $\mathcal{G}_g$ is allocated to DSPM at compile-time.

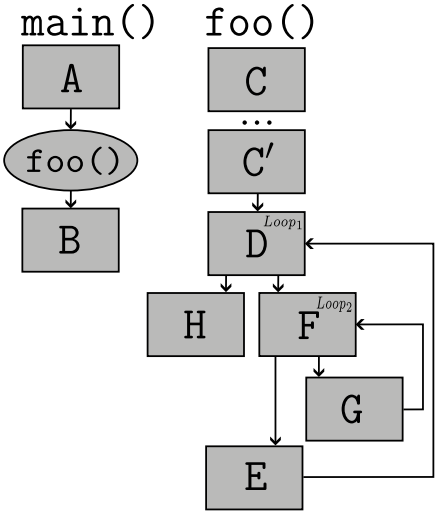
Listing 8.1: Exemplary program code

```
1  int a;
2
3  int main( void ) {
4    a = 100;
5    int b = foo( a );
6    return 0;
7  }
8
9  int foo( int a ) {
10   ...
11   int b = 0;
12   for( int i = 0; i <= a; ++i ) {
13     for( int j = 0; j <= i; ++j ) {
14       b++;
15     }
16     b += a;
17   }
18   return b;
19 }
```

Listing 8.1 provides a program code example with two functions (`main()` and `foo()`), two loops ($Loop_1$ and $Loop_2$), and one global data object (`int a`), and Figure 8.2 provides the resulting CFG of the exemplary program. In this example, $P = 2$, $M_1 = 1$, $M_2 = 3$, $\tilde{d} = 4$, and $d = 5$.

The function memory object `main()` consists of basic blocks `A` and `B`, and `foo()` consists of all remaining basic blocks from `C` to `H`. The loop memory

Figure 8.2.: CFG of the exemplary program code from Listing 8.1



object $Loop_1$ consists of `D` and `E`, and $Loop_2$ consists of `F` and `G`; each loop within a nested loop is considered a separate memory object. In function `foo()`, let us assume there are a bunch of basic blocks indicated by "..." between `C` and `C′`.

During dynamic allocation, the *entry point* function of the program is always placed in the Flash memory. The proper program execution requires a statically known start address of the *entry point*, so the *entry point* function itself can't be subject to dynamic allocation. However, no such constraint is applied on loops within the *entry point* function, except if the entry basic block of the *entry point* function is part of a loop. Therefore, we do not consider the *entry point* function of the program as a candidate for dynamic allocation, and we apply the following constraint to the decision variable vector x:

$$\begin{aligned} x_{(p,m)} = 0, \text{ if function } f_p \text{ is the } \textit{entry point } \wedge (m == 1 \vee \\ (m \neq 1 \wedge (b_r \in \mathcal{B}_{(p,m)} : b_r \text{ is entry basic block of } f_p))) \end{aligned} \tag{8.3}$$

$x_{(p,1)}$, i.e., $m == 1$, is the decision variable associated with the function memory object, and the *entry point* is not allocated dynamically to ISPM. In Figure 8.2, `main()` is the *entry point* of the program, and the `main()` function is not considered a dynamic allocation candidate.

$m \neq 1$ implies that the decision variables are associated with the loops within the function f_p. If the entry basic block of the *entry point* function is part of a loop memory object, that loop is also not allocated dynamically to ISPM. I.e., in Figure 8.2, if the entry basic block `A` in `main()` was part of a loop, then that loop would not be considered a dynamic allocation candidate either.

Moreover, if a function is allocated to ISPM, then the loops within that function are also allocated to ISPM. Therefore, we apply the following constraint to the decision variable vector x:

$$\forall m = \overline{2, M_p} \quad x_{(p,m)} = 1, \text{ if } x_{(p,1)} = 1 \tag{8.4}$$

I.e., if the p^{th} function is placed in ISPM, all the loops contained in the p^{th} function are also in ISPM—for all the values of m from 2 till M_p, $x_{(p,m)} = 1$. In Figure 8.2, if `foo()` is placed in the ISPM, the loop memory objects $Loop_1$ and $Loop_2$ must be placed in the ISPM.

As global data objects are allocated statically to DSPM according to the binary decision vector $x_{(\tilde{d}+1):d}$, the following DSPM size constraint is applied.

$$\sum_{g=(\tilde{d}+1)}^{d} S_{\mathcal{G}_g} x_g \leqslant S_{DSPM} \tag{8.5}$$

where $\forall g = \overline{(\tilde{d}+1), d}$, $S_{\mathcal{G}_g}$ is the size of the global data object $\mathcal{G}_g$, and S_{DSPM} is the total size of the DSPM.

At compile-time, the decision variable vector x decides which code object will be allocated to SPM during runtime. However, contrary to static allocation techniques, where the contents of the SPM remain the same throughout the execution, the contents within the SPM are dynamically replaced according to the decision variable vector x at runtime. During dynamic allocation, two memory objects that are simultaneously live—being executed or in use—cannot occupy the same address range in the SPM. Therefore, we need to perform a liveness analysis on the program to determine the live range of each memory object. The memory objects with conflicting live ranges cannot occupy the same address ranges in the SPM.

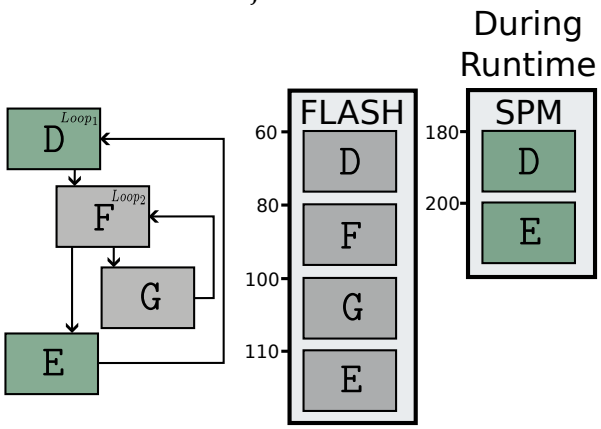
8.2.3 Address Objects

To dynamically allocate memory objects during runtime and to determine the SPM addresses to which each memory object is assigned, we need to solve an address assignment problem. A memory object might not be in consecutive memory addresses. For example, a loop within a nested loop could have some basic blocks in consecutive memory addresses, followed by another loop memory object, and then the remaining basic blocks—making it hard to assign just one SPM address to such memory objects. Therefore, we use address objects instead of memory objects during the address assignment algorithm.

Definition 8.2 (Address Objects)
An address object is defined as a set of basic blocks from the memory object that are in consecutive memory addresses within the Flash memory, i.e.,

$$\begin{aligned} \mathcal{A}_{(p,m,a)} = \{ & b_p^r \in \mathcal{M}_{(p,m)} \mid ((b_p^{(r+1)} \in \mathcal{M}_{(p,m)} \implies \alpha_p^r + \beta_p^r = \alpha_p^{(r+1)}) \vee \\ & (b_p^{(r-1)} \in \mathcal{M}_{(p,m)} \implies \alpha_p^r = \alpha_p^{(r-1)} + \beta_p^{(r-1)})) \vee \left|\mathcal{M}_{(p,m)}\right| = 1\}, \end{aligned} \tag{8.6}$$

Figure 8.3.: A nested loop example to show the difference between memory objects and address objects



- Only Loop_1 is allocated dynamically to SPM during runtime
- According to Listing 8.1 and Figure 8.2:
  - Loop_1 Memory Object: $\mathcal{M}_{(2,2)} = (\texttt{D}, \texttt{E})$
  - Loop_2 Memory Object: $\mathcal{M}_{(2,3)} = (\texttt{F}, \texttt{G})$
- Address objects for $\mathcal{M}_{(2,2)}$:
  - $\mathcal{A}_{(2,2,1)} = (\texttt{D})$
  - $\mathcal{A}_{(2,2,2)} = (\texttt{E})$
- Address object for $\mathcal{M}_{(2,3)}$:
  - $\mathcal{A}_{(2,3,1)} = (\texttt{F}, \texttt{G})$

where $a = \overline{1, A_{(p,m)}}$*, and* $A_{(p,m)}$ *is the total number of address objects associated with* $\mathcal{M}_{(p,m)}$*—the* m^{th} *memory object of the* p^{th} *function.* b_p^r *is the* r^{th} *basic block of the* p^{th} *function,* α_p^r *is the source address of* b_p^r *in the Flash memory, and* β_p^r *is the size of* b_p^r*.*

I.e., $\mathcal{A}_{(p,m,a)}$ is a set of basic blocks b_p^r associated with the memory object $\mathcal{M}_{(p,m)}$, such that the end address of the basic block b_p^r is the start address of the succeeding basic block $b_p^{(r+1)}$ or the start address of b_p^r is the end address of the preceding basic block $b_p^{(r-1)}$ in the Flash memory. If a memory object consists of a single basic block ($\left|\mathcal{M}_{(p,m)}\right| = 1$), the address object is equivalent to the memory object.

Figure 8.3 shows the nested loops from the example in Figure 8.2 and their placement in the Flash memory. In Figure 8.3, the end address of `D` is the start address of `F`, and the end address of `G` is the start address of `E` in the Flash memory. If only Loop_1 is allocated dynamically to the SPM, we need two

memory copy function calls to allocate address objects for D ($\mathcal{A}_{(2,2,1)}$) and E ($\mathcal{A}_{(2,2,2)}$) separately. We determine the set of address objects from memory objects and their respective memory addresses. If all basic blocks within the memory objects lie in the memory consecutively, then the memory object and address object are equivalent. For example, in Figure 8.3, $\mathcal{M}_{(2,3)}$ is equivalent to $\mathcal{A}_{(2,3,1)}$.

The address assignment algorithm uses first- and best-fit heuristics to determine the SPM address ranges for each address object such that no two address objects with conflicting live ranges occupy the same addresses in the SPM. For example, in Figure 8.3, if both Loop_1 and Loop_2 are dynamic allocation candidates, $\mathcal{A}_{(2,3,1)}$ has a liveness conflict with both $\mathcal{A}_{(2,2,1)}$ and $\mathcal{A}_{(2,2,2)}$. Therefore, we must allocate $\mathcal{A}_{(2,3,1)}$ to an address range that does not overlap with $\mathcal{A}_{(2,2,1)}$ and $\mathcal{A}_{(2,2,2)}$ in SPM.

8.2.4 Optimization Problem

In this subsection, we formulate a 2-dimensional compiler-level multi-objective dynamic SPM allocation-based optimization (MO_D) that aims to minimize the program's WCET and energy consumption simultaneously. The dynamic SPM allocation-based multi-objective optimization problem can be mathematically formulated as a minimization problem as follows:

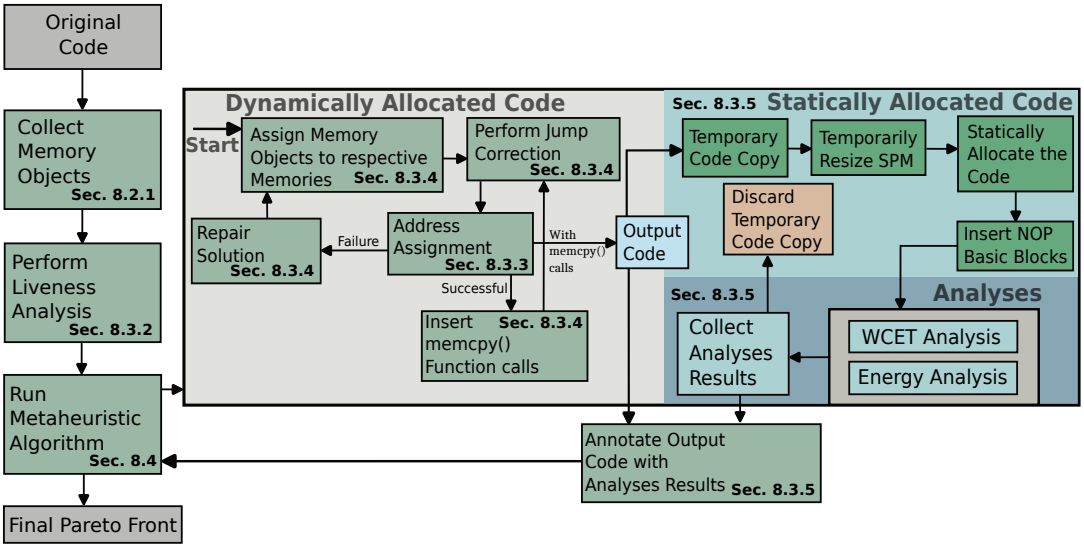
$$\begin{aligned} \min_{x} \quad & F(x) = (F_1(x), F_2(x)) \\ \text{subject to} \quad & \text{Eq. 8.3, Eq. 8.4, Eq. 8.5, and} (\mathbf{A} - \epsilon) = 0 \end{aligned} \tag{8.7}$$

where the objective function $F(x) \in \mathbb{R}^2$ represents the objective that needs to be minimized, $F_1(x)$ is the WCET objective, and $F_2(x)$ is the energy consumption objective corresponding to a solution vector x. $x \in X$ represents a d-dimensional binary decision variable vector, where Equation 8.1 and Equation 8.2 describe each coordinate $x_{(p,m)}$ and x_g, respectively, and $X \subset \{0,1\}^d$ is the search space of the multi-objective dynamic SPM allocation problem.

Equation 8.3 is the *entry point* constraint on the solution subvector $x_{(1,1):(P,M_P)}$. Equation 8.4 is the constraint to resolve conflicts between loops and their respective functions for the solution subvector $x_{(1,1):(P,M_P)}$. Equation 8.5 is the constraint for the static allocation of global data objects on the solution subvector $x_{(\tilde{d}+1):d}$. The fourth constraint $(\mathbf{A} - \epsilon) = 0$ implies that the address assignment algorithm presented in Subsection 8.3.3 should return 0 for the solution vector x, i.e., all the address objects must have a valid SPM address assigned by the algorithm. $\mathbf{A}$ is the total number of address objects that need dynamic allocation according to solution x, and ϵ is the total number of address objects that were actually assigned to the SPM after solving the address assignment problem.

The solution space of the dynamic SPM allocation problem consists of binary decision vectors $x \in X$ that fulfill Equations 8.3, 8.4, and 8.5 and occupy

Figure 8.4.: Multi-objective dynamic SPM allocation framework



space within SPM without violating any liveness constraints. While solving the proposed dynamic SPM allocation-based multi-objective optimization problem, we generate dynamically allocatable code for such decision vector x and analyze it to calculate the WCET and energy consumption of the code. In this chapter, the CPU copies code from the Flash to SPM and carries out the dynamic allocation of memory objects during runtime.

8.3 Dynamic SPM Allocation

In this section, we provide an overview of dynamic SPM allocation and introduce liveness analysis, address assignment algorithm, and generation and analysis of dynamically allocated code at the compiler level—that enables dynamic allocation of objects from one memory to another during runtime. This section is divided into the following subsections: Subsection 8.3.1 provides the context and steps required for the multi-objective dynamic SPM allocation and its overall picture within WCC. Subsections 8.3.2 and 8.3.3 discuss the liveness analysis and the address assignment algorithm, respectively, needed for dynamic SPM allocation. Subsections 8.3.4 and 8.3.5 present code generation and analyses of dynamically allocated code, respectively.

8.3.1 Contextual Overview

Figure 8.4 outlines the steps necessary to perform compiler-level multi-objective dynamic SPM allocation (MO_D). The steps for solving MO_D are proposed and described across Sections 8.3 and 8.4, and the sections and subsections associated with these steps are indicated within the figure. To generate dynamically allocatable code and solve the MO_D problem, firstly, we identify

and collect memory objects from the source code for dynamic allocation and perform liveness analysis (cf. Subsection 8.3.2) to determine the live ranges of the memory objects. Then, we call a metaheuristic algorithm (cf. Section 4.4) to solve the multi-objective optimization problem proposed and discussed in detail in Section 8.4. The metaheuristic algorithm generates binary solution vectors x that indicate which memory object will be dynamically allocated to which memory (cf. Subsection 8.2.1). To generate dynamically allocatable code, we assign memory objects to respective memories according to the solution vector x and perform appropriate code transformations (jump correction) to accommodate jumps between different memories (cf. Subsection 8.3.4).

We solve an address assignment problem (cf. Subsection 8.3.3) to find suitable memory addresses for address objects such that no liveness conflicts occur. If any address object is not assigned an SPM address, we repair the solution vector by assigning the associated memory object back to the Flash memory (cf. Subsection 8.3.4).

Once the address assignment problem is solved successfully, we insert a memory copy function ($memcpy()$) at the assembly level and $memcpy()$ function calls for each address object. The calls for the $memcpy()$ function initiate the copying of the address objects from the Flash memory to the SPM during runtime. Once we insert the $memcpy()$ function calls within the code, we again perform jump correction and solve the address assignment algorithm once more to accommodate the changes due to the $memcpy()$ function calls.

The optimized code containing the $memcpy()$ function calls that dynamically copy address objects at runtime is not usable for static WCET and energy analyses. To overcome this issue and to obtain proper WCET and energy objective values, this optimized code is converted into a temporary static code version by statically placing memory objects in SPM. It enables us to perform static WCET and energy analyses at compile time (cf. Subsection 8.3.5). Finally, we collect the analysis results, annotate them to the dynamic code, and discard the temporary static code. The analysis results are used as objective values while solving the multi-objective optimization using the metaheuristic algorithms.

8.3.2 Liveness Analysis

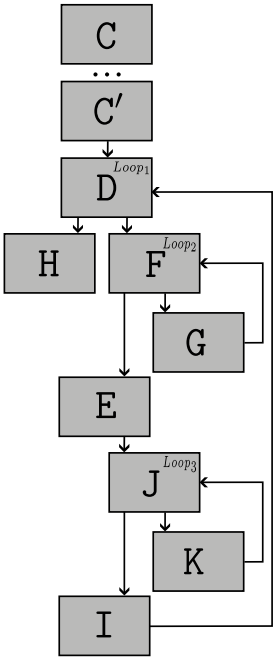
After determining memory objects within the program, a liveness analysis [App04] is performed within the compiler at the function level to determine the live ranges of memory objects. Algorithm 8.1 presents the steps for function- and loop-level liveness analysis performed within the WCC compiler.

The algorithm takes the set of memory objects (`MO`) as input and provides liveness data for all memory objects as output (Lines 1 and 2). Let $\mathcal{L}_{(p,m)}$ be the set of basic blocks that define the live range of $\mathcal{M}_{(p,m)}$, the m^{th} memory object in the p^{th} function. The liveness data consists of the following sets of basic blocks: *def*, *use*, and $\mathcal{L}$.

Algorithm 8.1 Function- and Loop-level Liveness Analysis

Input: Set of memory objects $\mathtt{MO}$
Output: Liveness data (*def*, *use*, and $\mathcal{L}$) for all memory objects
Perform Loop nesting forest analysis at the function level to get basic blocks contained within the loop and its entrance, exit, and nested loop information.
for $p = 1 : P$ **do** ▷ Iterate over all the functions
  for $m = 1 : M_p$ **do** ▷ Iterate over all the memory objects within the function f_p
    Initialize: $def_{(p,m)}$, $use_{(p,m)}$, and $\mathcal{L}_{(p,m)}$ as sets of basic blocks
    if $m == 1$ **then** ▷ The first memory object is the function itself
      for $p' = 1 : P$ such that $p' \neq p$ **do**
        Initialize: $\mathbb{B}$ - list of basic blocks calling function f_p; def', use', and $\mathcal{L}'$ - list of *def*, *use*, and $\mathcal{L}$ basic blocks for function $f_{p'}$
        for each basic block $b_{p'}$ in function $f_{p'}$ **do**
          if $b_{p'}$ calls function f_p **then**
            $\mathbb{B} \leftarrow b_{p'}$
            if $def' == \emptyset$ **then** ▷ The first $b_{p'}$ calling f_p is the *def* basic block
              $def' \leftarrow b_{p'}$
        for each basic block b_c in $\mathbb{B}$ **do**
          Set $\omega = \mathtt{false}$ - to check if b_c dominates any *def* basic blocks
          for each basic block b_d in def' **do**
            if $(b_c \neq b_d) \wedge (b_c \text{ dom } b_d)$ **then** ▷ Check if b_d dominates b_c
              $\omega = \mathtt{true}$
          if $\omega == \mathtt{false}$ **then**
            $def' \leftarrow b_c$ ▷ If b_c is not dominated by any *def*s, it is added to *def*
          else
            $use' \leftarrow b_c$ ▷ Else add it to *use* set
        for each b_d in def' **do** ▷ Iterate to determine the live range of f_p
          for each b_u in use' **do**
            if b_u dom b_c **then** ▷ Check if b_d dominates b_u
              $\mathcal{L}' \leftarrow \mathcal{L}' \cup \{\text{all basic blocks in between } b_c \text{ and } b_u\}$
        $def_{(p,m)} \leftarrow def'$, $use_{(p,m)} \leftarrow use'$, and $\mathcal{L}_{(p,m)} \leftarrow \mathcal{L}'$
          ▷ Collect *def*, *use*, and $\mathcal{L}$ sets from $f_{p'}$ associated with calls to f_p
    else ▷ Loops are live only within their own function
      for each b_p in function f_p **do** ▷ Iterate over all the basic blocks in f_p
        if b_p is the loop entrance basic block of $\mathcal{M}_{(p,m)}$ **then**
          if $\mathcal{M}_{(p,m)}$ is nested loop **then**
            $def_{(p,m)} \leftarrow$ {entrance basic block of the outermost loop}
          else ▷ Else loop's own entrance basic block is considered
            $def_{(p,m)} \leftarrow b_p$
        else if b_p is not the entrance basic block but a part of the loop **then**
          $use_{(p,m)} \leftarrow b_p$ ▷ The rest of the basic blocks are part of the *use* set
      $\mathcal{L}_{(p,m)} \leftarrow def_{(p,m)} \cup use_{(p,m)}$
        ▷ All basic blocks in *def* and *use* define loop's live range $\mathcal{L}$

Figure 8.5.: CFG of an exemplary program code with loop nests



In the case of a function memory object:

- *def* contains basic blocks that first call the function memory object $\mathcal{M}_{(p,1)}$ from function $f_{p'}$, where $\mathcal{M}_{(p,1)}$ and $f_{p'}$ are different functions. As mentioned in Section 8.2.1, for each $\mathcal{M}_{(p,m)}$, $m = 1$ implies that the memory object $\mathcal{M}_{(p,1)}$ is the function f_p itself.

- *use* contains subsequent basic blocks from $f_{p'}$ that call the function memory object $\mathcal{M}_{(p,m)}$.

- $\mathcal{L}_{(p,m)}$ contains all basic blocks from $f_{p'}$ that define the live range of the function memory objects $\mathcal{M}_{(p,m)}$ from the *def* basic block till the last *use* basic block within every function.

In Figure 8.2, `foo()` is called only once in `main()`, so the basic block `A` is its *def* basic block and its live range.

In the case of a loop memory object:

- *def* contains either the loop entrance or the entrance basic blocks of the outermost loop for the loop memory object $\mathcal{M}_{(p,m)}$, where $m \neq 1$.

- *use* contains all the subsequent basic blocks within the loop nest.

- $\mathcal{L}_{(p,m)}$ is a union of all basic blocks in *def* and *use*, as all the basic blocks within the loop nest define $\mathcal{M}_{(p,m)}$'s live range.

Each loop within a loop nest is considered a separate loop memory object. We perform a loop nesting forest analysis [Tar73] within the compiler at the function level to obtain basic blocks contained within every loop, along with their entrance, exit, and nested loop information (Line 3).

In Figure 8.2, D is the *def* basic block for Loop_1 and Loop_2, and D, E, F, and G define their live range. If the $memcpy()$ call for dynamically allocating Loop_2 is placed between D and F, then the $memcpy()$ call that transfers Loop_2 will be triggered on every iteration of Loop_1. Moreover, after the execution of Loop_2 from SPM is complete, basic block E from Loop_1 still needs to be executed. Due to such liveness conflicts, we cannot place Loop_2 and Loop_1 on overlapping SPM addresses. Consequently, there is no benefit to placing the $memcpy()$ function call for a loop memory object inside a loop nest. Therefore, even though basic block F is the loop entry of Loop_2, we dynamically allocate Loop_2 before entering Loop_1 to avoid iterative copying of Loop_2 and complicated liveness conflicts between loops within a loop nest, achieved by considering the loop entrance basic block of the outermost loop of the loop nest as the *def* basic block.

As we do not place the $memcpy()$ calls inside a loop nest in this thesis, we might encounter a limitation, as explained using the example in Figure 8.5. The loop memory object Loop_1 consists of basic blocks D, E, and I, whereas Loop_2 consists of F and G, and Loop_3 has basic blocks J and K. In this example, Loop_2 and Loop_3 are nested within Loop_1, and Loop_2 and Loop_3 are separate loops without any liveness conflicts. Now, we consider that only Loop_2 and Loop_3 are supposed to be dynamically allocated to SPM during runtime, and Loop_1 remains in the Flash memory. Then, according to our current approach, the $memcpy()$ function calls for Loop_2 and Loop_3 can only be placed between C′ and D, and Loop_2 and Loop_3 cannot have overlapping SPM address ranges because the $memcpy()$ function dynamically allocates both loops to SPM before either of their execution. However, there is no liveness conflict between Loop_2 and Loop_3. If both Loop_2 and Loop_3 cannot be placed simultaneously in SPM due to the SPM size constraint, then either of the loops is reassigned to the Flash using the repair mechanism (cf. Algorithm 8.3). If we allow the placement of the $memcpy()$ function calls inside a loop nest, such cases could be avoided, and Loop_2 and Loop_3 could be assigned overlapping SPM addresses. In the future, the liveness analysis and the dynamic memory allocation infrastructure within the WCC compiler could be expanded to accommodate such scenarios.

In Algorithm 8.1 during liveness analysis, we iterate over all the functions and all the memory objects within the function f_p, initialize $def_{(p,m)}$, $use_{(p,m)}$, and $\mathcal{L}_{(p,m)}$ sets for the current memory object $\mathcal{M}_{(p,m)}$, and check if $\mathcal{M}_{(p,m)}$ is a function or a loop (Lines 4-7). For functions, we iterate over all the functions except the current function f_p and check if any basic block calls f_p, and based on the first call or subsequent calls, we insert basic blocks in either $def_{(p,m)}$ or

$use_{(p,m)}$ and $\mathcal{L}_{(p,m)}$ sets (Lines 8-28). def', use', and $\mathcal{L}'$ contain lists of *def*, *use*, and $\mathcal{L}$ basic blocks associated with calls for function f_p in function $f_{p'}$. We initialize def', use', and $\mathcal{L}'$ for function $f_{p'}$. We also initialize a list $\mathbb{B}$ that will contain basic blocks that call function f_p from $f_{p'}$ (Line 9).

We iterate over all the basic blocks within $f_{p'}$ to collect all basic blocks that call function f_p and add them to $\mathbb{B}$. Moreover, the first basic block calling f_p from $f_{p'}$ is added to def' (Lines 10-14). During this iteration, def' contains only one basic block, i.e., the first basic block $b_{p'}$ that calls f_p (Line 14). However, more basic blocks might get added to def' if this $b_{p'}$ does not dominate the other basic blocks that call f_p from $f_{p'}$. For example, if a function call is within a branch of an if-else statement and this function call is never executed if the *def* basic block $b_{p'}$ is executed, then we need to consider such basic blocks as part of the *def* list too.

Therefore, we iterate over each basic block b_c from $\mathbb{B}$ that calls function f_p and check if b_c is dominated by any def' basic block (Lines 15-23). If the current basic block b_c that calls f_p is not dominated by any basic block in def', we need to add b_c to the def' list (Lines 17-21). If b_c is already dominated by a basic block in the def' list, then b_c is added to the use' list accordingly (Line 23).

After getting def' and use' lists for function $f_{p'}$, we insert all basic blocks in $\mathcal{L}'$ that define the live range of the function f_p in function $f_{p'}$ (Lines 24-27). Finally, def', use', and $\mathcal{L}'$ are added to $def_{(p,m)}$, $use_{(p,m)}$, and $\mathcal{L}_{(p,m)}$ (Line 28).

For loop memory objects, we iterate over all the basic blocks of the function f_p that contains the loop (Line 30). Depending on the results from the loop nesting forest analysis, we distinguish and insert basic blocks in $def_{(p,m)}$ and $use_{(p,m)}$ sets (Lines 31-37). If the loop is an inner loop in the loop nest, the entrance basic blocks of the outermost loop are considered part of $def_{(p,m)}$ (Lines 32 and 33). As the live range of a loop memory object is defined by all basic blocks associated with the loop nest, $\mathcal{L}_{(p,m)}$ is a union of $def_{(p,m)}$ and $use_{(p,m)}$ sets (Line 38).

8.3.3 Address Assignment Algorithm

Algorithm 8.2 presents the steps for solving the address assignment problem. Algorithm 8.2 takes the decision vector x and the set of address objects for all memory objects as input and provides the SPM address assignments for all address objects as output (Lines 1-2). Let $\mathbf{A}$ be the total number of address objects that need dynamic allocation according to solution x. After solving the address assignment problem, if all $\mathbf{A}$ address objects are assigned to SPM, the algorithm returns 0, and if not, $(\mathbf{A} - \epsilon)$, where ϵ is the total number of address objects that were actually assigned to the SPM. We initialize ϵ and the current ISPM size counter Γ terms to zero (Line 3). We use Γ to check the total size of the address objects already allocated to the ISPM.

With the help of the source address (α) in the Flash memory and the total size (β) for all the address objects, we solve the address assignment problem using

Algorithm 8.2 Address Assignment Algorithm

1: **Input:** Set of address objects and solution vector x
2: **Output:** SPM address γ for each address object and $(\mathbf{A} - \epsilon)$
3: **Initialize:** $\epsilon = 0$ and current ISPM size counter $\Gamma = 0$
4: **for** $p = 1 : P$ **do** ▷ Iterate over all the functions
5: **for** $m = 1 : M_p$ **do** ▷ Iterate over all the memory objects in the p^{th} function
6: **if** $x_{(p,m)} == 1$ **then** ▷ Check if $\mathcal{M}_{(p,m)}$ is allocated to SPM
7: **for** $a = 1 : A_{(p,m)}$ **do** ▷ Iterate over all the address objects in the m^{th} memory object
8: **if** $(\Gamma + \beta_{(p,m,a)}) \leqslant S_{ISPM}$ **then**
▷ First-fit heuristic is called only if SPM has the free capacity of $\beta_{(p,m,a)}$ size
9: Call first-fit heuristic to get SPM address $\gamma_{(p,m,a)}$
▷ First-fit heuristic assigns the first available SPM address to the address object
10: $\Gamma += \beta_{(p,m,a)}$ ▷ Increment Γ using the size of the current address object
11: $\epsilon += 1$ ▷ Increment ϵ to count the assigned address object
12: **else** ▷ If SPM is full, the best-fit heuristic is used to find the SPM address
13: Call best-fit heuristic to get SPM address $\gamma_{(p,m,a)}$
▷ Best-fit heuristic avoids liveness conflicts by fulfilling the constraint Equations 8.8 and 8.9
14: **if** $\gamma_{(p,m,a)}$ is a valid SPM address **then**
15: $\epsilon += 1$

the first-fit and best-fit heuristics to determine their destination addresses (γ) in SPM. We iterate over all the functions and the memory objects associated with the functions, and according to the solution x, if the considered memory object is supposed to be allocated to SPM, we try to find the SPM address for the address objects associated with the respective memory object (Lines 4-7). If the SPM has enough free capacity to accommodate the currently considered address object, we try to find a valid SPM address that can fit the considered address object (Lines 8-11).

The first-fit heuristic aims to fit as many address objects as possible within the SPM until it's full. If the SPM is full or does not have enough free capacity to accommodate the currently considered address object, we use the best-fit heuristic instead (Lines 13-15). The best-fit heuristic tries to find the best places in the SPM for the remaining address objects. Moreover, if any address objects are in consecutive memory addresses in the Flash (i.e., $\alpha_{(p,m,a)} + \beta_{(p,m,a)} = \alpha_{(p',m',a')}$), both first- and best-fit heuristics try to place them in consecutive addresses in SPM as well. If such address objects are consecutively placed in the SPM, we can refer to these multiple address objects as a single address object. In such a scenario, only one $memcpy()$ call can be used to transfer these address objects to SPM during runtime, reducing the number of required $memcpy()$ function calls.

We need to consider liveness conflicts when assigning overlapping SPM address ranges. We cannot assign two address objects associated with the same memory object to the same SPM address range, i.e., for example, in Figure 8.3, $\mathcal{A}_{(2,2,1)}$ and $\mathcal{A}_{(2,2,2)}$ cannot be assigned to the same SPM address range. Therefore, the following constraint condition is applied to the SPM addresses (γ) while using the best-fit heuristic:

$$\begin{aligned} \gamma_{(p,m,a)} + \beta_{(p,m,a)} \leqslant \gamma_{(p,m,a')} \vee \\ \gamma_{(p,m,a')} + \beta_{(p,m,a')} \leqslant \gamma_{(p,m,a)} \quad \forall a \neq a' \end{aligned} \tag{8.8}$$

Furthermore, address objects with overlapping executions (live range conflicts) are constrained to be allocated to different SPM address ranges during runtime, i.e.,

$$\begin{aligned} \gamma_{(p,m,a)} + \beta_{(p,m,a)} \leqslant \gamma_{(p',m',a')} \vee \\ \gamma_{(p',m',a')} + \beta_{(p',m',a')} \leqslant \gamma_{(p,m,a)} \text{ if } ((p',m') \neq (p,m)) \wedge \\ (\mathcal{A}_{(p,m,a)} \cap (\mathcal{A}_{(p',m',a')} \cup \mathcal{L}_{(p',m')}) \neq \emptyset) \end{aligned} \tag{8.9}$$

In this case, address objects associated with different memory objects are considered, i.e., for example, in Figure 8.3, if both $\mathcal{M}_{(2,2)}$ and $\mathcal{M}_{(2,3)}$ are allocated to SPM, $\mathcal{M}_{(2,2)}$ and $\mathcal{M}_{(2,3)}$ are live at the same time. Therefore, $\mathcal{A}_{(2,2,1)}$ and $\mathcal{A}_{(2,3,1)}$, as well as $\mathcal{A}_{(2,2,2)}$ and $\mathcal{A}_{(2,3,1)}$ cannot occupy the same SPM address range. For this reason, using the constraint Equation 8.9, the best-fit heuristic utilizes the liveness data such that no two simultaneously live memory objects are allocated to the same memory address range. The term $(\mathcal{A}_{(p,m,a)} \cap (\mathcal{A}_{(p',m',a')} \cup \mathcal{L}_{(p',m')})$ in Equation 8.9 reflects the live range conflicts between two address objects, $\mathcal{A}_{(p,m,a)}$ and $\mathcal{A}_{(p',m',a')}$. If $\mathcal{A}_{(p,m,a)}$ and the live range of $\mathcal{A}_{(p',m',a')}$ have any common basic blocks, the resulting set will be non-empty, and the constraint condition in Equation 8.9 will be applied.

8.3.4 Code Generation

Algorithm 8.3 presents the steps required to generate and analyze dynamically allocated code. As liveness analysis (cf. Algorithm 8.1) is independent of the solution vector x, we perform liveness analysis once in the very beginning, and the liveness results are valid for all possible solutions x (cf. Figure 8.4). Algorithm 8.3 takes the solution vector x, which decides if a memory object is allocated to SPM, as an input and outputs the dynamically allocated code that is analyzed using static WCET and energy analyzers (Lines 1 and 2). We statically allocate the memory objects to SPM or the Flash memory depending on the decisions in the solution vector x (Line 4). As the code needs to jump from the Flash to SPM and vice versa during runtime, we perform jump correction [OLF16] to replace the previously valid jumps to address objects in the Flash with new jumps to the respective runtime address objects in SPM.

Algorithm 8.3 Generation and analysis of the code for dynamic SPM allocation

1: **Input:** Solution vector x
2: **Output:** Dynamically allocated code and its analysis results
3: **do**
4: Statically allocate memory objects to SPM using x, perform jump correction, and reallocate memory objects back to the Flash
5: Perform address assignment using Algorithm 8.2
6: **if** $(\mathbf{A} - \epsilon) \neq 0$ **then**
7: Repair solution x
8: **while** $(\mathbf{A} - \epsilon) \neq 0$
9: Insert $memcpy()$ function in the Flash, insert $memcpy()$ function calls, and perform jump correction
10: Perform address assignment after code transformations
11: Generate a static version of the code and perform analyses

We statically allocate the code at this stage to SPM to perform jump correction. In Figure 8.2, if only $Loop_2$ is dynamically allocated to SPM, the previous jumps between `D` and `F` and `F` and `E` become invalid, so we need to fix jumps between these basic blocks using jump correction. Then, we re-allocate all the statically allocated code back to the Flash memory within WCC. These steps provide us with a jump corrected code in the Flash (Line 4).

Next, we solve the address assignment problem for the allocated code using Algorithm 8.2 (Line 5). It is important to note that we perform jump correction before solving the address assignment problem so that Algorithm 8.2 adjusts for the extra jump correction code too. Moreover, as we want to eventually insert $memcpy()$ function calls and perform some jump corrections (Line 9), it can increase the code size, and the second call to the address assignment algorithm might fail after this code transformation (Line 10). Therefore, we retrospectively consider a small tolerance for the SPM size while solving the address assignment problem the first time to compensate for the possible code size increase. If the address assignment algorithm does not assign some address objects to SPM, we repair the solution x by moving the memory objects associated with the unassigned address objects back to the Flash memory, i.e., we set their binary decision variable back to 0 (Lines 6 and 7). If the solution x is repaired, we need to again generate the dynamically allocated code by repeating steps 4 and 5 for the repaired solution (Line 8).

To copy address objects during runtime, we need the $memcpy()$ calls within the program that copy each address object of size β from its source address α from the Flash to its destination address γ, which is in the SPM. The values of α, β, and γ for all address objects are available after solving the address assignment problem (cf. Algorithm 8.2). Address objects are copied using the CPU—enabling dynamic allocation from α to γ during runtime. We insert the code for the $memcpy()$ function at the assembly level within the program

and place it within the Flash memory. We insert the $memcpy()$ function calls within the code and again perform jump correction to accommodate any broken jumps because of the newly added $memcpy()$ calls (Line 9). During runtime, the $memcpy()$ function is called, which copies the memory object from α in the Flash memory to γ in the SPM. Then, the jump-corrected code is executed, which enables the execution of the memory object from the destination address γ from the SPM instead of the source address α from the Flash memory.

After inserting $memcpy()$ function calls and performing jump correction (Line 9), the source addresses and the sizes of memory objects might change. Therefore, we again call the address assignment algorithm to get a new source address from the Flash (α), address object size (β), and destination address in SPM (γ) data that incorporates the $memcpy()$ calls and the jump correction information (Line 10).

8.3.5 Code Analyses

Finally, we need to perform WCET and energy analyses for multi-objective optimization. However, static analysis tools like `aiT` and `EnergyAnalyzer` cannot deal with memory objects dynamically copied from one memory to another. First of all, static analysis is unaware of the semantics of a dynamic copy, i.e., they fail to understand that a copied memory object at address γ behaves identically to the original memory object at address α, but just with different timings or energy properties due to the changed memories. Second, static analyzers fail to understand that the same SPM address γ could be occupied by different memory objects during a program's execution. To address these issues, we internally generate a temporary version of the SPM-allocated code only to enable static analysis.

In this temporary code version, all memory objects that should be copied dynamically onto the SPM are moved statically onto the SPM, and we substitute NOP basic blocks in the Flash memory in place of the statically allocated memory objects so that the Flash memory layout remains unchanged. The NOP basic blocks are never executed and do not contribute to the final WCET and energy results. These static duplicates are assigned to consecutive addresses within the SPM so that, within the temporary code, an SPM address γ is associated with at most one address object. If the size of the memory objects allocated to SPM exceeds the actual SPM size, the static analyzer assumes a larger SPM capacity to accommodate all statically allocated memory objects. The temporary code subject to static analysis contains all the $memcpy()$ function calls required for the actual dynamic SPM allocation so that all $memcpy()$-related overheads are reflected properly during static analysis. Moreover, the $memcpy()$ function is also part of the statically generated code. Therefore, the WCET and energy analyzers also account for the extra WCET and energy consumption due to the $memcpy()$ function.

We do not consider any involvement of caches within the underlying architecture. If caches are involved, dynamically changing the contents of the SPM can lead to issues with cache coherence. For real-time embedded systems, cache coherency must typically be ensured at the software level. This leads to the invalidation of the entire cache after a $memcpy()$ function dynamically allocates the memory object to the SPM. This invalidation results in a high number of cache misses when executing the allocated memory object from the SPM, worsening the WCET and energy consumption. Therefore, we could mark the address range of the SPM as non-cacheable to address this issue. However, this effectively renders the system as if no caches are present, aligning with the architecture assumed in this chapter.

Finally, the jumps within the temporary code version are corrected such that the execution flow through all memory objects and their static duplicates is identical to the original, dynamically allocated code. This way, all the dynamism present inherently in the final dynamically allocated code is reflected properly in the temporary code version, but in a purely static way. Moreover, as we can fully control the internal workings of the WCC compiler, no other compiler-level optimizations are carried out between analyses of the static version of the code and the generation of its dynamic version. Lastly, we use WCET and energy analyzers to perform respective analyses of this temporary static version of the code (Algorithm 8.3, Line 11). Once the static version of the code is analyzed, we collect the results and discard this temporary code version.

8.4 Multi-Objective Dynamic SPM Allocation-based Optimization

In this section, we exploit the metaheuristic algorithms to perform and solve the MO_D problem proposed in Equation 8.7. Figure 8.4 depicts the MO_D optimization framework, and Subsection 8.3.1 provides an overview of all the steps required to solve the MO_D problem within the compiler. The MO_D problem is solved using a metaheuristic algorithm (FPA or SPEA), and we obtain final solutions as a set of Pareto optimal solutions.

Algorithm 8.4 presents the dynamic SPM allocation-based multi-objective optimization performed at the compiler level. To initialize the algorithm, we recognize and collect all the memory objects by performing standard control flow and depth-first analyses within WCC (Line 1). Then, we perform the liveness analysis to determine the live ranges of the memory objects using Algorithm 8.1 (Line 2). To perform the multi-objective optimization, we need to initialize the metaheuristic algorithm. During evaluations, we use FPA and SPEA algorithms to solve the optimization problem. We perform evaluations using P different initial populations to reduce the influence of randomness on the final Pareto optimal solutions (Line 3). We randomly initialize the initial

Algorithm 8.4 Multi-objective dynamic SPM allocation (MO_D)

```
1:  Recognize and collect memory objects
2:  Perform Liveness Analysis using Algorithm 8.1
3:  for p = 1 : P do        ▷ Run metaheuristic algorithm for P different initial populations
4:      Randomly initialize the initial population of size N
5:      for n = 1 : N do
6:          Call Algorithm 8.3    ▷ Generate and analyze dynamically allocatable code for
                                    each individual in the initial population
7:      while Stopping criteria are not reached do
8:          Update all individuals in the current generation using a local
            or global update operator or both
9:          for Each updated Individual do
10:             Call Algorithm 8.3
11:         Update to the next generation using the selection operator
12:     Collect the Pareto optimal solution set
13: return Final Pareto optimal solution set (PF_F)
```

population of size N and then call Algorithm 8.3 to generate dynamically allocated code and analyze the individuals (Lines 4-6).

The code generated by Algorithm 8.3 consists of calls for the `memcpy()` functions that enable dynamic allocation of memory objects during runtime, required jump corrections to enable jumps between memories, and statically allocated data objects to DSPM. After performing analyses, we get WCET and energy consumption objectives for the dynamically allocated code by mimicking the dynamic behavior of the code statically at compile time, as explained in Subsection 8.3.5.

After initialization, we solve the MO_D problem by calling the metaheuristic algorithm (Lines 7-12). The metaheuristic algorithm uses local and global operators to update the individuals (Line 8). Each metaheuristic algorithm has its own local and global update operators—SPEA uses recombination and mutation operators, and FPA uses local pollination and global pollination operators (cf. Section 4.4). We generate code and perform WCET and energy evaluations for each updated individual using Algorithm 8.3 (Lines 9 and 10). Based on the WCET and energy objectives and Pareto dominance (cf. Definition 4.1), the metaheuristic algorithm uses a selection operator to select top-scoring individuals for the next generation (Line 11).

For the MO_D optimization problem, we set two stopping criteria—the maximum number of generations and the maximum number of generations for which the population remains the same (Line 7). Once one of the two stopping criteria is fulfilled, the algorithm provides the Pareto optimal solution set for the current initial population (Line 12). We collect all P Pareto fronts obtained after solving the MO_D optimization problem for each initial population and generate the final Pareto optimal solution set (PF_F) (Line 13).

8.5 Evaluation

In this section, we perform evaluations to compare the proposed MO_D optimization with the multi-objective static SPM allocation (MO_S) from Chapter 5 and the state-of-the-art ILP-based single-objective SPM overlay approach (SO_D) [VM07]. We implemented a version of the SO_D approach within WCC. As SO_D aims to minimize the energy objective, we perform WCET analysis for their final solution to obtain respective energy values. In the case of SO_D, we always have one solution on its final Pareto front as it is a single-objective optimization problem. Moreover, we compare MO_D solutions with the baseline evaluation BL, where the whole code is executed from the Flash.

Subsection 8.5.1 describes the evaluation setup used to solve the optimization problems. Subsection 8.5.2 presents the Pareto optimal solutions found after solving MO_D, MO_S, SO_D, and BL. Subsection 8.5.3 compares the quality of MO_D, MO_S, SO_D, and BL solutions using the quality indicators described in Subsection 4.3.3. Subsection 8.5.4 presents compilation time results for MO_D, MO_S, and SO_D optimization runs.

8.5.1 Evaluation Setup

Similar to previous chapters, the evaluations are conducted using the ARM Cortex-M0 fork of the WCC compiler. We use the memory subsystem with separate ISPM and DSPM, as shown in Figure 8.1, for the STM32F0xx implementation of ARM Cortex-M0 [STM22a]. The code is generated for each benchmark by the WCC compiler using the `-O2` optimization flag. The static WCET and energy analyses for MO_D, MO_S, SO_D, and BL are performed using `aiT v21.04i` and `EnergyAnalyzer v21.04i`, respectively. We solve the multi-objective MO_D and MO_S optimizations using FPA and SPEA algorithms and the single-objective SO_D optimization using the ILP solver `Gurobi` [Gur24].

We evaluated 20 benchmarks from `AutoBench`, `Networking`, and `TeleBench` benchmark suites offered by the `Embedded Microprocessor Benchmark Consortium` (`EEMBC`) [PCLGO09] during evaluations. `EEMBC` offers benchmarks for embedded systems that model different state-of-the-art, complex, real-world applications. `AutoBench` has automotive and industrial benchmarks, `Networking` has benchmarks associated with moving packets in networking applications beneficial for routers and switches, and `TeleBench` has modem and fixed-telecom-related benchmarks.

We adjust the ISPM and DSPM sizes individually to 60% relative to the benchmark size to increase the pressure on the optimization. Moreover, we assume that the benchmarks will fit within the Flash memory. As the random initial population can influence the final Pareto front of the metaheuristic algorithm, we repeat the evaluations for both FPA and SPEA algorithms using $P = 5$ different initial populations. All the individual Pareto fronts from the P optimization runs for P initial populations are combined to find the final

Pareto front (PF_F) (cf. Algorithm 8.4). We set a timeout value of 200 h for each benchmark, and WCC terminates the compilation if the optimizations are not completed successfully. I.e., we treat the timeout value as another stopping criterion for the metaheuristic algorithm. Within this time limit, WCC evaluates the benchmarks using both FPA and SPEA algorithms, and for each of the algorithms, WCC repeats the evaluations $P = 5$ times. If the task set runs into a timeout, we collect the approximated Pareto solutions from the final population of the algorithm as the final Pareto output.

The SO_D optimization run requires two energy evaluations to build the ILP model—one energy evaluation where all the code is placed in SPM and another energy evaluation where all the code is placed in the Flash memory. We perform a WCET and energy analysis after solving the ILP problem. Given that we need to solve the ILP problem once and perform in total four evaluations to solve the SO_D optimization problem, we set a time limit of 2 h on the SO_D optimization run. We expect to finish the SO_D optimization faster compared to MO_D and MO_S as long as the ILP solver can solve the optimization problem.

The MO_D and MO_S optimization problem, while using FPA or SPEA, requires a WCET and energy analysis for each individual (2 evaluations). For FPA and SPEA algorithms, the population size is 10, the maximum number of generations is 50, and the number of initial populations is $P = 5$. Therefore, we need to perform a maximum of $2 * 10 * 50 * 5 = 5000$ evaluations to finish the optimization for each metaheuristic algorithm. As we need $2 * 5000 = 10000$ evaluations to optimize each benchmark using both FPA and SPEA algorithms, and the benchmarks in `EEMBC` are larger than MRTC benchmarks used in previous chapters, we set the time limit of 200 h on the MO_D and MO_S optimization runs. If the metaheuristic algorithm reaches the second stopping criterion—the maximum number of generations with the same population (cf. Tables 5.1 and 5.2)—before meeting the first—the maximum number of generations—, the optimization might require fewer than 10000 evaluations.

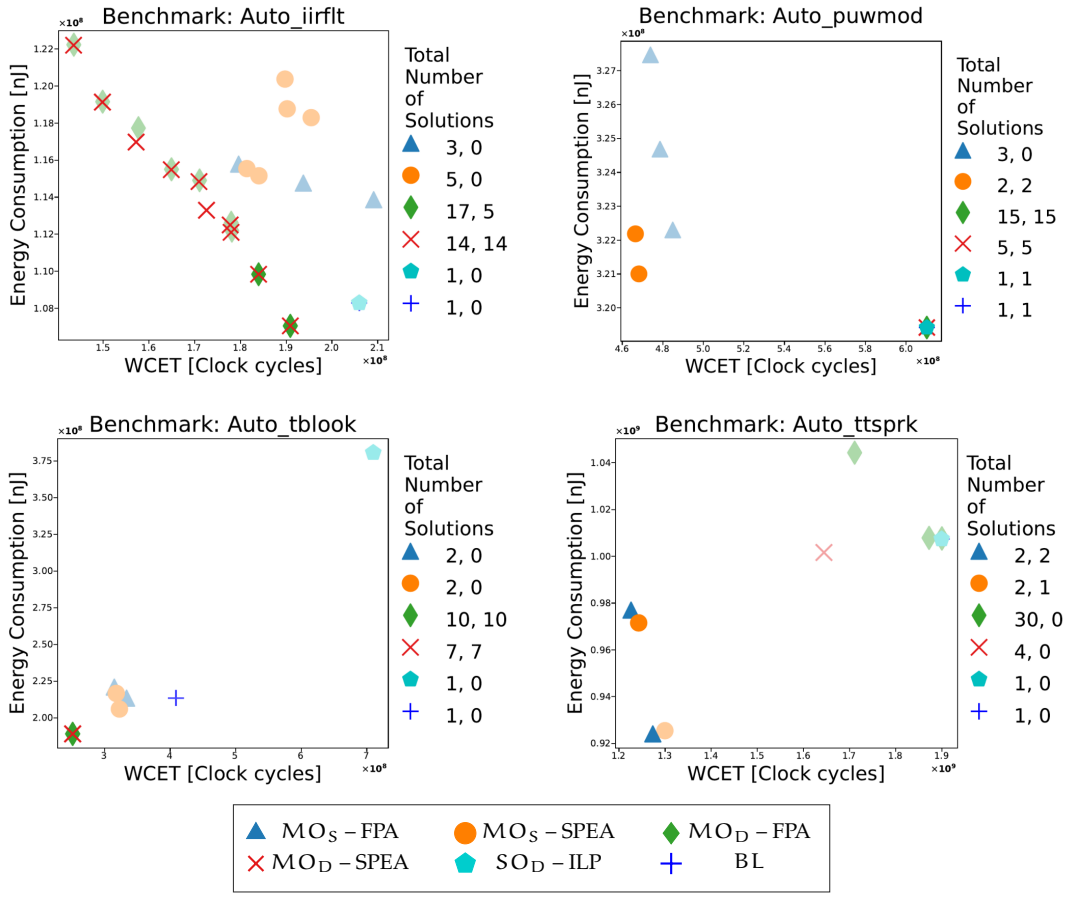
The values of the hyperparameters for FPA and SPEA are depicted in Tables 5.1 and 5.2, respectively. The results obtained during these evaluations are valid for the algorithm parameters described in the respective tables.

8.5.2 Pareto Fronts

In this subsection, we compare all the solutions obtained after performing MO_D, MO_S, SO_D, and BL evaluation runs. Figure 8.6 presents the solutions found by MO_D and MO_S for both FPA and SPEA algorithms, and SO_D and BL have only one solution each on their respective fronts. For brevity's sake, we present the solutions for only four benchmarks in this subsection. The figures for all the remaining benchmarks are in Appendix D.

The final Pareto fronts in Figure 8.6 are represented using a 2D scatter plot for the following four benchmarks: `Auto_iirflt`, `Auto_puwmod`, `Auto_tblook`, and `Auto_ttsprk`. The x-axis and the y-axis represent WCET and energy

Figure 8.6.: Pareto fronts for four benchmarks obtained using MO_S, MO_D, SO_D, and BL evaluation runs



consumption objectives, respectively. The title of each plot is the name of the benchmark from the respective benchmark suites from `EEMBC`. The legend at the bottom of the figure represents the optimization runs to which the solutions belong.

Moreover, the legend on the right side of each subfigure shows the total number of solutions returned by respective optimization runs on their respective Pareto fronts (PF_F). The same legend indicates the number of solutions on the final Pareto front (PF_T) out of the total solutions on their PF_F. The final Pareto front (PF_T) represents the set of indifferent solutions from the union of the Pareto optimal solutions (PF_F) obtained from all the approaches. The darker-colored solutions in the figure represent the solutions on PF_T, and the fainter version of those colors represents the solutions on the respective PF_Fs. As the optimization problems considered in this chapter are minimization problems, the solutions in the bottom-left corner of the subfigures generally represent the final Pareto front.

For the `Auto_iirflt` benchmark, the final PF_T consists of solutions only contributed by MO_D–FPA and MO_D–SPEA. SO_D was not able to find any

better solution than the baseline BL solution. MO_S–FPA and MO_S–SPEA found 3 and 5 solutions on their respective PF_Fs. However, none of those solutions are on the final PF_T. Moreover, we can also see that one of the MO_S–FPA solutions is worse than the BL solution in terms of WCET and energy consumption. As the MO_S–FPA's PF_F consists of a solution worse than BL, we can deduce that this MO_S–FPA evaluation run never encountered an individual with all basic blocks in the Flash memory (the BL condition).

As MO_S and MO_D do not have a constraint that eliminates solutions worse than the BL, we can encounter such scenarios. Therefore, we can extend the MO_S and MO_D optimizations by adding a constraint that removes the solutions where the objective values are worse than BL, avoiding such cases.

In the case of the `Auto_puwmod` benchmark, except for MO_S–FPA, all approaches (MO_S–SPEA, MO_D–FPA, MO_D–SPEA, SO_D, and BL) contributed to the final PF_T. The objective values of the solutions contributed by MO_D–FPA, MO_D–SPEA, and SO_D are equal to the BL's objectives. However, all these solutions are different in the solution space. All these solutions place various combinations of data objects in DSPM, but none of the functions or loops are dynamically allocated to the ISPM. The archived populations during the optimization show that if any functions or loops were assigned to the ISPM, the objective values were higher than the objective values of solutions on their respective PF_Fs. Moreover, the static allocation of any global data object to DSPM does not show any change in the benchmark's WCET and energy consumption. Consequently, we received multiple solutions converging to the same point in the objective space. Therefore, as mentioned in Section 7.7, if we could identify such unnecessary decision variables beforehand, we could reduce the solution space of the optimization problem.

Even after placing memory objects within ISPM, the WCET and energy consumption of a solution can still increase. We need to add the extra jump correction code that enables jumping from the Flash to SPM. Moreover, for MO_D, we need to add `memcpy()` function calls that dynamically allocate the memory objects to ISPM during runtime. The static analyzers consider the impact of the `memcpy()` function on the program's WCET and energy consumption during analysis. Therefore, we can encounter cases like the `Auto_puwmod` benchmark when the extra WCET and energy cost due to jump correction and `memcpy()` function is higher than the benefits of placing the memory object in ISPM.

For `Auto_tblook`, only MO_D–FPA and MO_D–SPEA contributed to the final PF_T. `Auto_tblook` has 4 functions, 8 loops, and 18 global data objects. For all the MO_D solutions on PF_T, 0 functions and the same 6 loops out of 8 were placed in ISPM. However, all the solutions are different in the solution space due to the placement of different global data objects in DSPM. However, similar to the `Auto_puwmod` case, none of the global data objects had any impact on the objective values. Therefore, all the MO_D solutions converged to a single point in the objective space. Moreover, SO_D contributed a solution that is worse

Table 8.1.: Comparison between the average number of solutions (in %) on the final PF_T from the respective PF_F of each optimization approach

MO_S		MO_D		SO_D	BL
FPA	SPEA	FPA	SPEA	ILP	
18.03	50.82	25.93	40	10	10

than the BL solution. The SO_D solution placed two out of 4 functions within ISPM. However, the `memcpy()` function contributed 54.18% and 56.29% extra WCET and energy consumption to its objective values, leading to such high final WCET and energy consumption values.

In the case of the `Auto_ttsprk` benchmark, only MO_S contributed to the final PF_T. Neither SO_D nor MO_D was able to find a single solution that is better than any of the MO_S solutions.

From overall evaluations, we calculated the total number of solutions provided by each optimization run on the final PF_T and averaged it over all the benchmarks. Table 8.1 provides the percentage of the number of solutions on the final Pareto front PF_T, on average, out of the number of solutions on the respective optimization's PF_F. From the table, we can see that MO_S–SPEA contributed a higher average percent of solutions to the final PF_T from their PF_Fs. However, both MO_D–SPEA and MO_D–FPA provided more solutions on the final PF_T compared to MO_S–FPA.

Moreover, SO_D and BL provided the least number of solutions from their PF_F on the final PF_T. In the case of `Auto_a2time` and `Auto_puwmod`, none of the optimizations (MO_D, SO_D, and MO_S) were able to find any solutions that have both objectives lower than the BL's objectives. Therefore, the BL solution lies on the final PF_T in these cases.

As mentioned in Sections 5.4.2 and 6.5.3, Table 8.1 can skew numbers towards single-solution-producing approaches or approaches with a lower total number of solutions on their PF_Fs. For example, SO_D and BL both contributed solutions to the final PF_T for only 2 benchmarks out of 20, leading to a 10% contribution to PF_Ts from their respective PF_Fs. Similarly, even though MO_S–SPEA contributed a higher percentage of solutions to PF_T from its PF_F, MO_D–FPA and MO_D–SPEA contributed a higher total number of solutions to the final PF_T.

From the Pareto fronts of all the benchmarks, we can see that MO_D performed better than MO_S–FPA, and MO_S–SPEA performed better than MO_D. Moreover, both MO_D and MO_S are better than the state-of-the-art SO_D optimization.

The performance of MO_D can vary depending on the size of the functions and loops compared to the MO_S, which is a basic block-level optimization. We can assume that the size of functions and loops will almost always be greater than the basic blocks contained within them—except where functions and

Table 8.2.: Average reduction in WCET and energy consumption objectives (in %) obtained using MO_S, MO_D, and SO_D over BL

Objectives	MO_S		MO_D		SO_D
	FPA	SPEA	FPA	SPEA	ILP
WCET	38.36	38.58	36.7	36.74	6.37
Energy consumption	14	14.34	11.26	11.49	-0.65

loops have only one basic block. If a benchmark contains functions or loops larger than the size of the ISPM, then they are directly disqualified as dynamic allocation candidates. However, MO_S can still place basic blocks from such functions or loops within ISPM.

We observed that all the considered benchmarks from `EEMBC` have a large function, "`t_run_test()`," which is called from the benchmark's `main()` function. The `t_run_test()` function does not fit in the ISPM in our 60% relative SPM size case, i.e., this function is bigger than 60% of the total size of the benchmark. Moreover, for most of the considered benchmarks, `t_run_test()` consists of a large nested loop, which likewise does not fit within the ISPM. Consequently, such large functions are instantly disqualified by MO_D, and only smaller functions and loops that can fit in ISPM are considered dynamic allocation candidates. Therefore, a system designer can choose to use MO_D if the following condition is satisfied:

$$\sum_{p=1}^{P} \sum_{\substack{m=1 \\ \forall S_{\mathcal{M}_{(p,m)}} \leqslant S_{ISPM}}}^{M_p} S_{\mathcal{M}_{(p,m)}} \geqslant S_{ISPM} \tag{8.10}$$

where $S_{\mathcal{M}_{(p,m)}}$ is the size of the memory object $\mathcal{M}_{(p,m)}$, and S_{ISPM} is the size of the ISPM. Equation 8.10 implies that the sum of the sizes of the memory objects smaller than ISPM size must be greater than or equal to ISPM size. If this condition is not satisfied, MO_S might provide better-quality solutions.

We also compared the reduction in WCET and energy consumption objectives achieved using MO_D, MO_S, and SO_D over the baseline BL evaluation. Table 8.2 reports the percentage reduction in the objective values obtained using the optimization approaches over BL. In Table 8.2, percentage reductions are computed as the reduction of average objective values across all benchmarks, relative to the BL evaluations. From Table 8.2, we observe that the reduction in WCET is higher than that in energy consumption for all approaches. MO_S provided a slightly higher reduction of average objective values compared to MO_D, whereas both MO_D and MO_S achieved larger reductions in WCET and energy objectives than SO_D.

For some benchmarks, SO_D generated solutions with worse WCET and energy consumption than BL. This can be due to the extra WCET and energy

Table 8.3.: Total number of benchmarks with best (or nondominated) quality indicators for each optimization approach

Quality Indicators	MO_S		MO_D		SO_D	BL
	FPA	SPEA	FPA	SPEA	ILP	
$\mathcal{C}$	2 (2)	7 (4)	0 (7)	1 (7)	1 (1)	0 (0)
NR	2 (1)	8 (1)	3 (2)	3 (2)	1 (0)	0 (0)
RNS	2 (2)	7 (4)	0 (7)	1 (7)	1 (1)	0 (0)
HV	3 (1)	9 (1)	0 (5)	1 (5)	1 (0)	0 (0)

consumption required to copy memory objects from one memory to another using the $memcpy()$ function. We could avoid this situation by constraining the SO_D's final solution using the BL solution, i.e., if SO_D's final solution has higher objective values than BL, we could discard that solution and output the original solution. Consequently, in Table 8.2, we see a slight increase in average energy consumption due to SO_D compared to BL.

As explained before, we need to perform jump correction and add extra $memcpy()$ code to facilitate the dynamic allocation of memory objects during runtime. The previous jumps, branches, and load/store instructions can go out of range after adding this extra code, requiring extra code transformations to fix the control flow of the program and leading to an increase in code size. We averaged the code size of all the MO_D solutions for all the benchmarks and compared it with the BL solution's code size. MO_D caused, on average, a 52.76% increase in code size compared to the original benchmark size.

Moreover, we calculated the WCET and energy overheads that the MO_D solutions incur due to the dynamic allocation of memory objects during runtime using $memcpy()$. For all benchmarks, $memcpy()$ functions contributed 24.39% and 22.65% to the total WCET and energy consumption, respectively. We can see that the dynamic copying of code from one memory to another using the $memcpy()$ function contributes to the overall degradation of the solution quality of the MO_D optimization runs. To further analyze these trends, we evaluate these solutions using quality indicators in the next subsection.

8.5.3 Quality Indicators

To evaluate and compare the quality of the proposed MO_D, we use four quality indicators, namely *Coverage* ($\mathcal{C}$) (cf. Definition 4.6), *Non-dominance Ratio* (NR) (cf. Definition 4.7), *Ratio of Non-dominated Solutions* (RNS) (cf. Definition 4.8), and *HyperVolume* (HV) (cf. Definition 4.9). As the exact Pareto front is unknown, we consider the final Pareto front PF_T, the best available approximation of the exact Pareto front, to calculate the quality indicators. For reference sake, Tables D.1 and D.2 in Appendix D present the quality indicator values for all the benchmarks from these evaluations.

Table 8.3 summarizes the total number of benchmarks with best or non-dominated $\mathcal{C}$, NR, RNS, and HV indicators for each optimization approach. The numbers shown outside the bracket in this table are the total number of benchmarks with best quality indicators, and the numbers inside the bracket represent the number of benchmarks that obtained non-dominated Pareto optimal solutions using multiple optimization techniques.

For example, for the Auto_tblook benchmark (cf. Figure 8.6, Table D.1, and Table D.2), MO_D–FPA and MO_D–SPEA contributed all the solutions from their PF_F to the final PF_T. So, these two techniques have the same $\mathcal{C}$ and RNS metric values. Moreover, all these solutions converge to the same point in the objective space, leading to the same HV metric values. Therefore, for Auto_tblook, the performance of MO_D–FPA and MO_D–SPEA is indifferent in terms of $\mathcal{C}$, RNS, and HV indicators—the benchmarks with such quality indicators are quantified in the table by the value inside the bracket. However, as MO_D–FPA provided the largest number of solutions to the final PF_T compared to MO_D–SPEA, the performance of MO_D–FPA is the best in terms of NR indicator for this benchmark—the benchmarks with such best quality indicators are quantified in the table by the value outside the bracket.

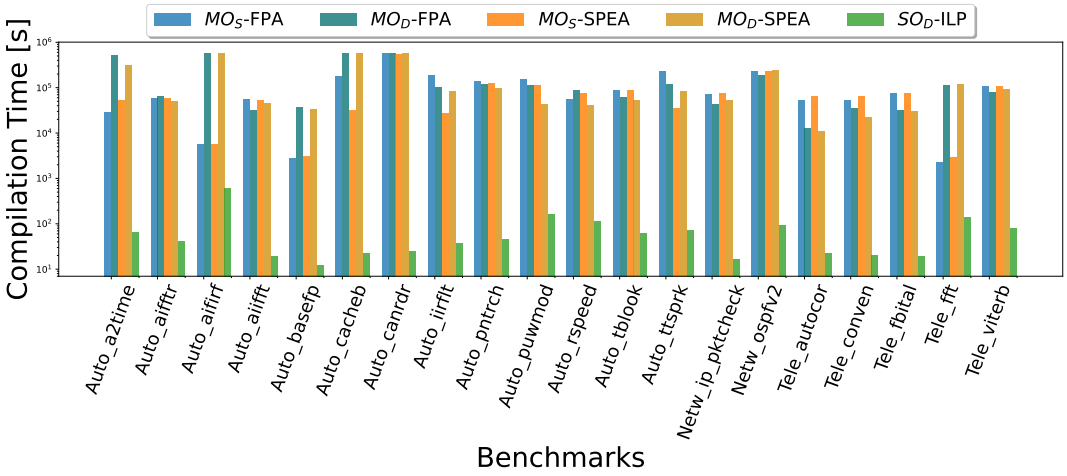
Table 8.3 shows that, on average, MO_S–SPEA provided better quality solutions for more benchmarks than the other approaches. However, MO_D–FPA and MO_D–SPEA provided better quality solutions than MO_S–FPA. From overall evaluations, we observed that MO_D provided a larger number of solutions on the final PF_T than MO_S. Nevertheless, the quality of the MO_D solutions was hampered due to excessive $memcpy()$-related objective costs. In any case, MO_D and MO_S performed better than the state-of-the-art SO_D optimization.

8.5.4 Compilation Times

In this subsection, we compare the compilation time required to solve the MO_D, MO_S, and SO_D optimizations with each other. Figure 8.7 presents the compilation time required by MO_D, MO_S, and SO_D in the form of a bar plot. The x-axis represents all the evaluated benchmarks from EEMBC, and the y-axis shows the compilation time required to finish the optimizations. From the figure, we can see that MO_D and MO_S were more time-consuming compared to single-objective SO_D during this evaluation. SO_D took, on average, 26.08 min to finish the optimization; MO_S–FPA and MO_S–SPEA took 52.68 h and 45.48 h, on average, to finish the optimization runs, respectively; MO_D–FPA and MO_D–SPEA took 48.89 h and 44.07 h, on average, to finish the optimization runs, respectively.

As mentioned in Subsection 8.5.1, we need to perform at most 10000 evaluations to solve MO_D and MO_S optimization problems using FPA and SPEA. The code generation and analysis phase of the MO_D problem is more time-consuming compared to the MO_S problem, as we need to perform

Figure 8.7.: Compilation times for MO_S, MO_D, and SO_D optimization runs



liveness analysis and solve the address assignment problem at least twice for each individual. However, MO_D took, on average, 5.6% less time than MO_S optimization runs.

As the solution space of MO_D is defined by the total number of functions, loops, and global data objects, and the solution space of MO_S is defined by the total number of basic blocks, it might happen that the solution space of the MO_S problem is bigger than the MO_D problem, and the second stopping criteria is hit first by the MO_D optimization run. Therefore, we observed that for some benchmarks, MO_D was faster than the MO_S optimization run during this evaluation. On the contrary, from Figure 8.7, we can also see that MO_S required less compilation time than MO_D for some benchmarks.

8.6 Conclusion

In this chapter, we proposed a novel compiler-level dynamic SPM allocation-based multi-objective optimization (MO_D) that simultaneously minimizes the WCET and energy consumption of a program. The dynamic SPM allocation model proposed in this chapter handles the dynamic movement of memory objects from the Flash memory to SPM. Moreover, we extended the WCET and energy consumption analyses framework within the WCC compiler to handle analyses of such dynamically allocated code.

To solve the dynamic SPM allocation-based multi-objective optimization problem, we used two metaheuristic algorithms—FPA and SPEA. During evaluations, we evaluated and compared the results of the proposed MO_D optimization with MO_S and SO_D using real-world benchmark suites from `EEMBC`. Evaluations showed that the MO_S–SPEA optimization run provided better solutions for more benchmarks than MO_D. However, MO_D performed better than MO_S–FPA and provided more solutions on the final Pareto front

than both MO_S–FPA and MO_S–SPEA. Moreover, MO_D and MO_S clearly outperformed SO_D.

The dynamically allocatable code uses the CPU to copy code from the Flash memory to SPM during runtime. Therefore, the WCET and energy consumption required by the $memcpy()$ function degrade the quality of solutions compared to the solutions contributed by MO_S. Moreover, dynamic allocation requires extra $memcpy()$ function calls and extra jump correction code, increasing the code size. During these MO_D evaluation runs, we observed, on average, a 52.76% increase in code size compared to the original benchmark size. Furthermore, the $memcpy()$ functions contributed, on average, 24.39% and 22.65% to the WCET and energy consumption of the benchmarks, respectively.

This chapter is the first step toward compiler-level dynamic SPM allocation-based multi-objective optimization. However, we need to improve the approach proposed in this chapter by reducing WCET, energy consumption, and code size overheads. In this chapter, we saw that the overheads in terms of WCET and energy consumption due to $memcpy()$ are significant. Therefore, in the next chapter, we explore methods to decrease the overhead incurred due to $memcpy()$ and improve the MO_D solution quality. We integrate the Direct Memory Access (DMA) controller within WCC to use it in conjunction with the dynamic SPM allocation model for copying code dynamically during runtime.

Furthermore, in Chapter 10, we integrate code compression into the multi-objective dynamic SPM allocation technique to simultaneously minimize WCET, energy consumption, and code size of dynamically allocatable code.

9

DMA-aware Multi-Objective Dynamic SPM Allocation

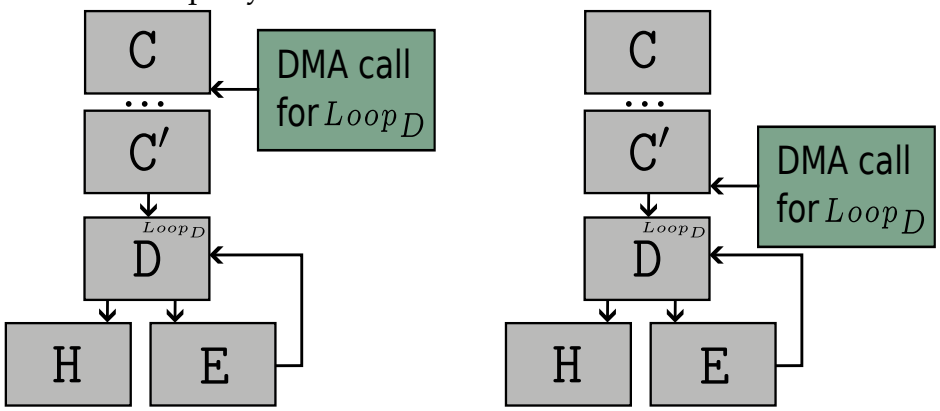
In Chapter 8, we exploited compiler-level dynamic SPM allocation to perform multi-objective optimization and minimize the program's WCET and energy consumption simultaneously. Dynamic memory allocation enables copying objects on and off memory during runtime, but presents some challenges. We observed during evaluations in the previous chapter that the extra WCET and energy costs incurred due to the *memcpy()* function were very high (24.39% and 22.65%, respectively).

The *memcpy()* function interrupts the program's execution and uses necessary CPU resources to copy address objects from one memory to another. Interrupting the CPU to copy functions and loops from the Flash memory to SPM during runtime can add further WCET and energy overheads. This interference can degrade the code quality and inversely impact the aim of optimizing the code. As mentioned in Section 2.4.3, a Direct Memory Access (DMA) controller can transfer code from one memory to another, freeing the CPU to perform timing-critical executions. Therefore, we propose a compiler-level DMA model and an accompanying DMA analyzer in this chapter. We also propose a DMA-aware dynamic SPM allocation-based multi-objective optimization (MO_{DMA}). MO_{DMA} aims to allocate functions and loops to SPM during runtime using DMA and minimize the program's WCET and energy. We solve MO_{DMA} using two metaheuristic algorithms—FPA and SPEA.

The DMA controller must finish transferring before the respective function or loop needs execution, or the CPU stalls until the pending transfer is complete. We can reduce such CPU stalls by strategically placing DMA calls within the code, ensuring that transfers are completed before the execution reaches the respective function or loop. Figure 9.1 provides an example CFG with a loop $Loop_D$, where $Loop_D$ consists of two basic blocks, `D` and `E`. Let us assume that there are a bunch of basic blocks indicated by "..." between basic blocks `C` and `C'`. Figure 9.1 **(a)** shows that a DMA transfer call for $Loop_D$ is placed after basic block `C`, and Figure 9.1 **(b)** shows that the DMA transfer call is placed after `C'`.

In Figure 9.1 **(a)**, the $Loop_D$'s DMA transfer is initiated after the execution of `C`. The DMA controller transfers $Loop_D$ to SPM in arbitration mode (cf. Section 2.4.3) whenever it receives access to the bus matrix while the CPU

Figure 9.1.: Exemplary CFG with DMA transfer call



(a) DMA transfer call after C **(b)** DMA transfer call after C′

executes all the basic blocks until C′. To ensure that the Loop_D's DMA transfer is complete before executing Loop_D, we use the polling mechanism to check if its DMA transfer is complete (cf. Section 2.4.3). In Figure 9.1 **(b)**, the DMA transfer call is placed immediately before Loop_D's entrance basic block. Consequently, the CPU is stalled, and the DMA transfer occurs in burst mode.

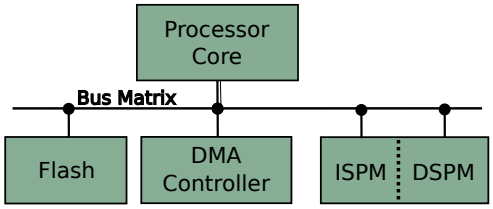
The extra time the DMA controller requires to transfer code from one memory to another depends on the placement of DMA transfer calls within the program. Depending on the DMA transfer mode, the extra time required to perform the DMA transfer varies, i.e.,

- If the DMA transfer occurs in burst mode, the extra time is equal to the execution time the DMA controller takes to perform the DMA transfer in isolation. This scenario is comparable to the use of the $memcpy()$ function by the CPU in Chapter 8.

- If the DMA transfer occurs in arbitration mode, the extra time required is equal to the total time the CPU is stalled while the DMA controller has access to the bus matrix.

For brevity's sake, we refer to this extra time as the *total execution time cost* required for the DMA transfer. Therefore, there are two total execution time costs for the DMA transfer: one for burst mode ($\mathcal{T}(x)$) and another for arbitration mode ($\mathcal{T}'(y)$). x is a binary decision variable that decides which memory object will be dynamically allocated to the ISPM during runtime and which global data object will be statically allocated to the DSPM (cf. Section 8.2.1). y is a binary decision variable that determines whether a DMA call will be placed after a particular basic block. We present detailed explanations for $\mathcal{T}(x)$ and $\mathcal{T}'(y)$ in the upcoming sections. If only burst mode is considered, the total execution time cost ($\mathcal{T}(x)$) depends only on the address object size transferred to the SPM. It does not depend on where we place the

Figure 9.2.: Exemplary memory hierarchy with DMA



DMA transfer call in the program. Therefore, as the total execution time cost in burst mode $\mathcal{T}(x)$ is constant, we have no optimization potential for $\mathcal{T}(x)$.

However, the total execution time cost in arbitration mode $\mathcal{T}'(y)$ depends on where we place DMA transfer calls within the program. Therefore, we propose a novel single-objective DMA Call Placement Optimization (DCPO), which aims to find optimal places within the code to place DMA transfer calls. DCPO strives to minimize the total execution time cost required by the DMA controller to perform dynamic allocation during runtime. As ILPs are better suited to solve single-objective optimizations and can provide optimal solutions, we opted to solve the DCPO optimization using the ILP solver `Gurobi` [Gur24]. During evaluations, we also solve the DCPO problem using FPA and SPEA algorithms to compare the results obtained after solving the ILP-based DCPO problem and determine the best method to solve DCPO.

Figure 9.2 shows an exemplary memory hierarchy with a DMA controller used in this chapter. The underlying architecture for dynamic SPM allocation consists of a Flash memory, an instruction SPM (ISPM), and a data SPM (DSPM). During evaluations, we generate code, perform analyses, and perform compiler-level MO_{DMA} optimization for ARM Cortex-M0. Figure 9.2 exactly represents the STM32F0xx implementation of the ARM Cortex-M0 architecture [STM22a, STM22b]. The WCET and energy analyses performed during the evaluations also consider underlying timing and energy models for the ARM-Cortex-M0 architecture. We can effortlessly expand the DMA-aware multi-objective dynamic SPM allocation proposed in this chapter to support DRAM, SRAM, or both in addition to Flash, as long as the timing and energy characterization of DRAM, SRAM, or both are available.

Typically, commercial DMA controllers are equipped with multiple DMA channels that can be programmed for simultaneous data transfers. However, in this chapter, we focus solely on memory-to-memory DMA transfers from Flash to SPM, where both Flash and SPM are single-ported memories. If we consider multiple DMA channels working simultaneously, we introduce more conflicts between each DMA channel competing for memory access, leading to more overhead and pessimism in the DMA model. As we want to model the execution time cost needed to perform DMA transfers at compile time as precisely as possible, we consider a single DMA transfer at a time in this chapter. Moreover, for STM32F0xx, only one DMA channel can be programmed to perform memory-to-memory transfer at a time.

The DMA-aware multi-objective dynamic SPM allocation proposed in this chapter was published in Springer's International Journal of Time-Critical Computing Systems (Real-Time Systems) in 2025 [JF25]. The key contributions of this chapter are:

- We introduce a compiler-level DMA model and a DMA analyzer and integrate it within the WCC compiler. In contrast to previous works, our DMA model assumes single-ported memories and can account for blocking times due to resource conflicts when both the CPU and the DMA controller access memories simultaneously. Still, our DMA model is flexible enough to also support architectures with dual-ported memories, which has been assumed by previous works in such cases, to the best of our knowledge.
- We propose a single-objective DCPO optimization to find optimal places within the code to place DMA calls and solve it using ILPs, FPA, and SPEA.
- We estimate WCET and energy consumption of the DMA-aware dynamically allocatable code as precisely as possible at compile-time.
- We propose DMA-aware dynamic SPM allocation-based multi-objective optimization (MO_{DMA}) to minimize the program's WCET and energy consumption.
- We solve MO_{DMA} using two metaheuristic algorithms—FPA and SPEA.

During evaluations, we compare the proposed MO_{DMA} with state-of-the-art single-objective ILP-based dynamic memory allocation with and without DMA and with multi-objective static and dynamic SPM allocation. In Chapter 8, we compared the proposed MO_D optimization with the MO_S approach from Chapter 5 and a state-of-the-art ILP-based single-objective SPM overlay optimization (SO_D) proposed by Verma et al. [VM07]. In this chapter, we compare MO_D, MO_S, and SO_D approaches with the proposed MO_{DMA} optimization approach to determine the effectiveness of DMA-aware dynamic allocation over the previous optimization techniques. We also compare the proposed MO_{DMA} optimization with the state-of-the-art DMA-aware ILP-based single-objective optimization [KBS17] (hereafter referred to as SO_{DMA}). Moreover, we compare MO_{DMA} with a baseline (BL) evaluation, where the whole program is executed from the Flash memory. The key results from the evaluations are:

- DCPO–ILP outperformed both DCPO–FPA and DCPO–SPEA across all benchmarks.
- DCPO–FPA performed worse than both DCPO–ILP and DCPO–SPEA.

- On average, MO_{DMA} provided higher quality solutions, surpassing all previously proposed and state-of-the-art approaches.
- MO_{DMA}–FPA and MO_{DMA}–SPEA using DCPO–ILP, on average, provided 86.75% and 78.57% solutions on the final Pareto front.
- We achieved a 19.17% and 7.64% average reduction in WCET and energy consumption, respectively, using MO_{DMA}+DCPO–ILP compared to the multi-objective dynamic SPM allocation that uses the CPU instead for dynamically allocating code during runtime (MO_D).
- MO_{DMA}–FPA+DCPO–ILP achieved, on average, 37.81% and 11.82% reduction in WCET and energy consumption compared to the BL evaluation run.

The chapter is organized as follows: Section 9.1 provides an overview of the related background work. Section 9.2 presents some preliminary information related to dynamic SPM allocation, DCPO, and mathematical formulation for the MO_{DMA} problem. It also presents the DMA-aware code generation and analysis. Section 9.3 presents the proposed compiler-level DMA model, DMA analysis, and DCPO. Section 9.4 presents the metaheuristic algorithm-based MO_{DMA} optimization. Section 9.5 presents the evaluation results, and Section 9.6 concludes the chapter.

9.1 Related Work

Some research has been done to exploit a DMA controller to perform single-objective compiler-level optimizations. Udayakumaran et al. [UB03] proposed a compiler-level dynamic allocation method for global and stack data, which dynamically allocates data using DMA to reduce a program's runtime.

Soliman et al. [SP17] presented a compiler-directed prefetching scheme to dynamically allocate global data and stack objects using a DMA controller to optimize the program's WCET. They converted large local objects to global objects to reduce the maximum stack size so that they could consider the rest of the function stack as a whole stack object for dynamic allocation. They assumed that the data SPM is dual-ported—allowing the CPU to access the data SPM and simultaneously transfer the data between SPM and main memory using the DMA controller. Therefore, the overheads when the CPU and the DMA controller try to access the data SPM simultaneously are not considered—restricting their approach for dual-ported SPM cases.

Similarly, Francesco et al. [FMA+04] used a dual-ported SPM and a dedicated DMA engine to perform runtime SPM management. Dasygenis et al. [DBD+06] proposed an approach that used a DMA controller and software prefetching mechanisms to hide memory latency and get performance and energy improvements. Yang et al. [YWY+10] presented a data pipelining

technique that uses a DMA controller to dynamically allocate data to SPM during runtime. They also used a dual-ported SPM model to overlap the CPU execution and DMA operations. To the best of our knowledge, the existing works are overly optimistic due to their assumption of multi-port memories. As a consequence, these existing approaches are not applicable to simple, low-cost, and single-ported architectures.

Kim et al. [KBS17] proposed a DMA-aware ILP-based single-objective optimization that performs dynamic SPM allocation (SO_{DMA}). However, the SO_{DMA} approach performs dynamic allocation of only functions and aims to minimize only WCET as an objective. They performed dynamic allocation of complete functions using the DMA controller without interruptions in the burst mode, during which CPU execution is halted.

Contrary to MO_{DMA}, these approaches focus on optimizing only one objective. Moreover, we do not assume a dual-ported SPM. Therefore, we might encounter cases where the DMA controller and the CPU need to access the same memory, and one of them is halted by the bus matrix—the overheads due to such halts are considered by the proposed DMA analyzer. Furthermore, unlike previous approaches, our DMA model can easily be used to analyze dual-ported systems—consider the overheads due to such halts to be zero.

9.2 Fundamentals of Dynamic SPM Allocation using DMA

In this chapter, we use DMA transfer calls instead of the $\mathtt{memcpy}()$ function calls to perform dynamic allocation during runtime. This section discusses the dynamic allocation preliminaries and the generation and analysis of DMA-aware dynamically allocatable code. It is divided into the following subsections: Subsection 9.2.1 provides preliminary information regarding dynamic SPM allocation, DCPO, and MO_{DMA}. It also presents the mathematical formulation for the MO_{DMA} optimization problem. Subsection 9.2.2 provides the context and steps required for DMA-aware multi-objective dynamic SPM allocation and its overall picture within the compiler. Subsection 9.2.3 presents the steps for code generation and analysis of DMA-aware dynamically allocatable code.

9.2.1 Problem Formulation

Similar to Chapter 8, in this chapter, we consider the dynamic allocation of memory objects (cf. Definition 8.1) from the Flash to ISPM, where $\mathcal{M}_{(p,m)}$ is the m^{th} memory object in the p^{th} function. Moreover, $x_{(p,1)}$ is the decision variable for the p^{th} function itself, and the rest of the decision variables ($x_{(p,2):(p,M_p)}$) are decisions for the loops within that function. We also consider the static allocation of global data objects ($\mathcal{G}_g$) from the Flash to DSPM using startup code, where $\mathcal{G}_g$ is the g^{th} global data object, which is live throughout the

execution of the program. Moreover, as a memory object might not be in consecutive memory addresses, we use address objects (cf. Definition 8.2) instead during dynamic allocation, where $\mathcal{A}_{(p,m,a)}$ is the a^{th} address object associated with the m^{th} memory object in the p^{th} function. In this chapter, we use calls to a DMA controller to dynamically allocate address objects instead of engaging the CPU with the unnecessary task of copying during runtime.

$x = (x_{(1,1)}, .., x_{(1,M_1)}, .., x_{(P,M_P)}, x_{(\tilde{d}+1)}, .., x_d) \in X$ is a d-dimensional binary decision variable vector. $X \subset \{0,1\}^d$ is the search space for the DMA-aware multi-objective dynamic SPM allocation, P is the total number of functions in the program, M_p is the total number of memory objects in the p^{th} function, and $\tilde{d}$ is the total number of memory objects within the program. The subvector $x_{(1,1):(P,M_P)}$ represents the dynamic allocation decision vector for memory objects, and $x_{(p,m)} \in \{0,1\}$ decides if a memory object will be dynamically allocated to ISPM or not during runtime. The subvector $x_{(\tilde{d}+1):d}$ represents the static allocation decision vector for global data objects, and $x_g \in \{0,1\}$ decides if a global data object will be statically allocated to DSPM or not.

We apply the same three constraints discussed in Section 8.2.2 to the binary decision variable vector x while solving the DMA-aware multi-objective dynamic SPM allocation problem in this chapter.

1. The first constraint, Equation 8.3, is applied to the *entry point* function and its entry basic block if the entry basic block is part of a loop memory object.

2. The second constraint, Equation 8.4, is applied to the loops within a function if the function is dynamically allocated to the ISPM.

3. The third constraint, Equation 8.5, is a DSPM size constraint applied to the global data objects.

In this chapter, we use a DMA controller to dynamically move address objects from the Flash memory to the SPM during runtime. Our goal is to identify appropriate places within the program to insert function calls for the DMA transfer of address objects, ensuring that these transfers are completed before the address objects are executed from the SPM. To solve this problem, we propose a DMA Call Placement Optimization (DCPO), which aims to find optimal places within the program to place DMA transfer calls (cf. Section 9.3). Let $\delta_{(p,m,a)}$ be the DMA transfer call for the address object $\mathcal{A}_{(p,m,a)}$. Exactly one DMA call is needed to transfer one address object.

To place the DMA call $\delta_{(p,m,a)}$ for $\mathcal{A}_{(p,m,a)}$ within the p^{th} function f_p, DCPO uses a binary decision variable vector $y_{(p,m,a)} \in \{0,1\}^{(B_p-1)}$, where B_p is the total number of basic blocks in f_p. y is the overall binary decision variable vector that decides where DMA transfer calls will be placed within the program. The total execution time cost for all DMA transfers in arbitration mode ($\mathcal{T}'(y)$) is calculated for the solution vector y (cf. Section 9.3). The proposed DCPO

minimizes $\mathcal{T}'(y)$ and aims to find the optimal solution y, which dictates the placement of the DMA calls within the program. For each solution vector x that decides which memory object is allocated to SPM during runtime, we solve the DCPO optimization problem and get the appropriate DMA call placements.

The solution space of the dynamic SPM allocation problem consists of binary decision vectors $x \in X$ that fulfill Equations 8.3, 8.4, and 8.5 and occupy space within ISPM without violating any liveness constraints. While performing the proposed MO_{DMA} optimization in this chapter, we generate the DMA-aware dynamically allocatable code for such a decision vector x and analyze it to calculate the WCET and energy consumption of the code. However, due to the inherent dynamic nature of DMA transfers, static analysis cannot capture the execution time and energy consumption required for DMA transfers.

Furthermore, the energy model used by the `EnergyAnalyzer` described by Equation 2.4 in Section 2.2 does not account for the energy consumption of the DMA controller. Therefore, the tools used are not able to account for the energy costs incurred due to DMA arbitration and CPU stalls during runtime. However, we estimate the energy costs incurred due to the extra memory accesses during DMA transfers. We add this extra cost ($\mathcal{E}(x)$) to the energy consumption value computed by the `EnergyAnalyzer` to compensate for the lack of consideration of the DMA controller in the energy model. Consequently, with an extended energy model that considers the energy consumption due to a DMA controller, we can get more accurate energy consumption estimates.

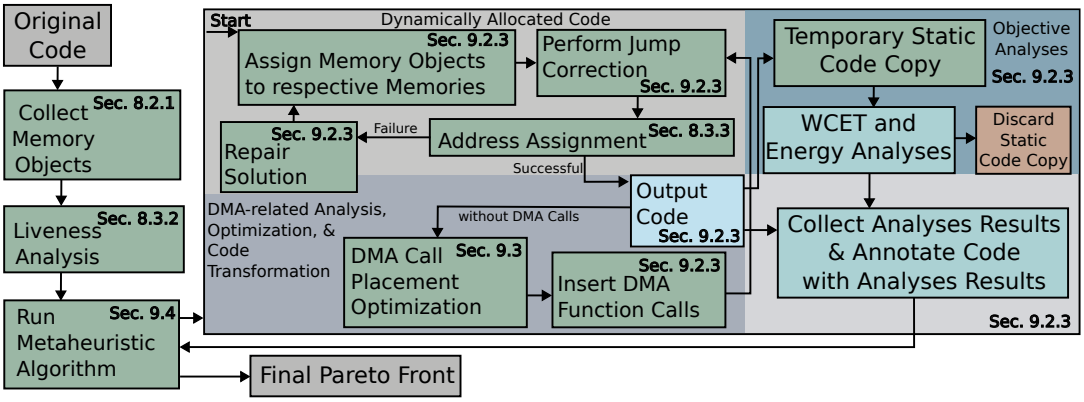
Moreover, in terms of extra execution time, we already calculated this value ($\mathcal{T}'(y)$) by solving the DCPO problem. Therefore, WCET+$\mathcal{T}'(y)$ and energy consumption+$\mathcal{E}(x)$ are considered the minimization objectives for the MO_{DMA} problem. The proposed MO_{DMA} aims to minimize WCET+$\mathcal{T}'(y)$ and energy consumption+$\mathcal{E}(x)$ simultaneously and tries to find optimal solutions $x \in X$, which dictates the dynamic allocation of memory objects during runtime.

The following is the mathematical formulation for the multi-objective dynamic SPM allocation optimization problem.

$$\begin{aligned} \min_{x} \quad & F(x) = ((F_1(x) + \mathcal{T}'(y)), (F_2(x) + \mathcal{E}(x))) \\ \text{subject to} \quad & \text{Eq. 8.3, Eq. 8.4, Eq. 8.5, and} (\mathbf{A} - \epsilon) = 0 \end{aligned} \tag{9.1}$$

where objective functions $F_1(x) \in \mathbb{R}$ and $F_2(x) \in \mathbb{R}$ represent WCET and energy consumption corresponding to a solution vector $x \in X$. The program's WCET and energy consumption are calculated in WCC using the static analyzer tools `aiT` and `EnergyAnalyzer`. These static analyzers provide WCET and energy consumption values of the complete program, i.e., $F_1(x)$ and $F_2(x)$. The energy objective $F_2(x)$ equates to the energy consumption calculated by the `EnergyAnalyzer` using the energy model described by Equation 2.4.

Figure 9.3.: DMA-aware multi-objective dynamic SPM allocation



$$\mathcal{E}(x) = \sum_{p=1}^{P} \sum_{m=1}^{M_p} \left(\mathbb{F}_p^r \sum_{a=1}^{A_{(p,m)}} (e_F + e_S)\beta_{(p,m,a)} \right) x_{(p,m)} \tag{9.2}$$

$\mathcal{E}(x)$ is the extra energy consumption cost incurred due to the extra memory accesses during the DMA transfers of the address objects. e_F is the energy cost for reading from the Flash memory, and e_S is the energy cost for writing to the SPM. From Equation 2.4, we set the values of e_F and e_S as: $e_F = 1.037625441$ nJ and $e_S = 1.031341343$ nJ. $\beta_{(p,m,a)}$ is the size of the address object $\mathcal{A}_{(p,m,a)}$ that is being dynamically allocated to the SPM, and $\mathbb{F}_p^r$ is the execution frequency of the basic block b_p^r, after which the DMA call for $\mathcal{A}_{(p,m,a)}$ is placed by DCPO.

The value of $\mathcal{T}'(y)$ is obtained from the DCPO solved in Section 9.3. Equations 8.3 and 8.4 are constraints on the solution subvector $x_{(1,1):(P,M_P)}$ for the dynamic allocation of memory objects. Equation 8.5 is a constraint on the solution subvector $x_{(\tilde{d}+1):d}$ for the static allocation of global data objects. As mentioned in Subsection 8.3.3, $\mathbf{A}$ is the total number of address objects that need dynamic allocation according to solution x, and ϵ is the total number of address objects that were actually assigned to the SPM after solving the address assignment problem. The fourth constraint $(\mathbf{A} - \epsilon) = 0$ says that the address assignment algorithm should return 0 for the solution x.

9.2.2 Contextual Overview

Figure 9.3 outlines the steps necessary to perform compiler-level DMA-aware multi-objective dynamic SPM allocation (MO_{DMA}) optimization. The steps for solving the MO_{DMA} problem are proposed and elaborated across Sections 9.2, 9.3, and 9.4 in this chapter and Sections 8.2 and 8.3 in Chapter 8. The sections associated with these steps are indicated within the figure. To generate dynamically allocatable code and solve the MO_{DMA} problem, firstly, we identify and collect memory objects from the source code for dynamic allocation. We then perform liveness analysis using Algorithm 8.1 to determine

the live ranges of the memory objects. The function-level liveness analysis performed within the compiler is described in detail in Subsection 8.3.2 in Chapter 8.

Then, we call a metaheuristic algorithm (cf. Section 4.4) to solve the proposed multi-objective optimization problem (cf. Equation 9.1) in Section 9.4. The metaheuristic algorithm generates binary solution vectors x that indicate which memory object will be dynamically allocated to which memory. To generate dynamically allocatable code, we assign the memory objects to respective memories according to the solution vector x and perform appropriate code transformations (jump correction) to accommodate jumps between different memories (cf. Subsection 9.2.3).

We solve the address assignment problem using Algorithm 8.2 to find suitable memory addresses for address objects such that no liveness conflicts occur. The address assignment algorithm is described in detail in Subsection 8.3.3 in Chapter 8. If any address object is not assigned an SPM address, we repair the solution vector by assigning the associated memory object back to the Flash (cf. Subsection 9.2.3).

Once the address assignment problem is solved, we perform a DMA Call Placement Optimization (DCPO) (proposed in Section 9.3) for the solution vector x. The goal of DCPO is to minimize $\mathcal{T}'(y)$, where y is the binary decision variable that determines whether a DMA call will be placed after a particular basic block. The final solution vector y (with minimized $\mathcal{T}'(y)$) provides appropriate places within the code to insert DMA calls that enable dynamic code allocation during runtime.

The files containing the architecture-specific DMA controller-related functions are part of the WCC compiler. The compiler places the DMA transfer function in the Flash memory and makes it available for our use at the assembly level. The architecture-specific DMA transfer function is generally internally configured to manage polling or interrupt mechanisms. For STM32F0xx, we use the Hardware Abstraction Layer (HAL) code within WCC to support DMA transfers during runtime. However, during WCET and energy analyses, these HAL-related functions are treated as external functions within WCC, and their WCET and energy contributions are assumed to be zero. After we insert the DMA function calls within the program, we again perform jump correction and re-solve the address assignment problem to accommodate the changes due to DMA calls.

The optimized code containing DMA calls that dynamically copy memory objects at runtime is not usable for static WCET and energy analyses. To overcome this issue and to obtain proper WCET and energy objective values, this optimized code is converted into a temporary static code version by statically placing memory objects in SPM. It enables us to perform static WCET and energy analyses at compile time (cf. Subsection 9.2.3). Finally, we collect the analysis results, annotate them to the dynamic code, and discard the

Algorithm 9.1 Generation and Analysis of the code for dynamic SPM allocation

```
1: Input: Solution vector x
2: Output: Dynamically allocated code and its analysis results
3: do
4:     Allocate memory objects to SPM using x and perform jump correction
5:     Perform address assignment using Algorithm 8.2
6:     if (A − ϵ) ≠ 0 then
7:         Repair solution x
8: while (A − ϵ) ≠ 0
9: Solve the DCPO problem
10: Insert DMA transfer calls and perform jump correction
11: Perform address assignment after code transformations
12: Generate a static version of the code and perform analysis
```

temporary static code. The analysis results are used as objective values while solving the multi-objective optimization using the metaheuristic algorithms.

9.2.3 Code Generation and Analyses

Algorithm 9.1 presents the steps required to generate and analyze dynamically allocated code. As liveness analysis (cf. Algorithm 8.1) is independent of the solution vector x, we perform liveness analysis once in the very beginning, and the liveness results are valid for all possible solutions x (cf. Figure 9.3). Algorithm 9.1 takes the solution vector x, which decides if a memory object is allocated to SPM, as an input and outputs the dynamically allocated code that is analyzed using static WCET and energy analyzers (Lines 1 and 2). We allocate the memory objects to SPM or the Flash memory depending on the decisions in the solution vector x (Line 4). As we want the code to jump from the Flash memory to SPM and vice versa during runtime, we perform jump correction [OLF16] to replace the previously valid jumps to address objects in the Flash memory with new jumps to the respective runtime address objects in SPM (Line 4). As mentioned in Section 8.3.4, we perform jump correction before solving the address assignment problem so that the address assignment algorithm (cf. Algorithm 8.2) accounts for the extra jump correction code as well.

Next, we solve the address assignment problem for the allocated code using Algorithm 8.2 (Line 5). As we want to eventually insert DMA transfer calls within the code (Line 10), this can increase the code size, and the second call to the address assignment algorithm might fail after this code transformation (Line 11). Therefore, we retrospectively consider a small tolerance for the SPM size while solving the address assignment problem. If the address assignment algorithm does not assign some address objects to SPM, we repair solution x by moving the memory objects associated with the unassigned address objects

back to the Flash memory. I.e., we set their binary decision variable back to 0 (Lines 6 and 7). If the solution vector x is repaired, we need to again generate the dynamically allocated code by repeating steps 4 and 5 for the repaired solution (Line 8).

To copy address objects during runtime, we need DMA calls within the program that copy each address object of size β from its source address α from the Flash memory to its destination address γ, which is in the SPM. The values of α, β, and γ for all address objects are available after solving the address assignment problem (cf. Algorithm 8.2). Address objects are copied using the DMA controller—enabling dynamic allocation from α to γ during runtime.

Then, we solve the DCPO optimization problem (cf. Section 9.3), which provides us with appropriate places within the code to insert DMA calls (Line 9). We insert DMA calls within the code and again perform jump correction to accommodate any broken jumps because of newly added DMA calls (Line 10). We again call the address assignment algorithm to get a new source address (α), address object size (β), and destination address (γ) data that incorporates the DMA calls and the jump correction information (Line 11).

Similar to Section 8.3.5, we need to generate a static version of the dynamically allocatable code to enable static WCET and energy analyses at the compiler level. The dynamic allocation code in this chapter consists of DMA function calls instead of $memcpy()$ function calls. The temporary code subject to static analysis contains all the DMA calls required for the actual dynamic SPM allocation so that all DMA-related overheads are reflected properly during static analysis. The temporary code version properly reflects the dynamism present inherently in the final DMA allocated code, but in a purely static way.

However, the extra costs incurred due to DMA arbitration and CPU stalls during runtime are not considered by a static analyzer at compile time. Therefore, we use the DMA analyzer proposed in Section 9.3 to calculate these extra costs in terms of total execution time cost ($\mathcal{T}'(y)$) and add them to the statically analyzed WCET value. Consequently, we can guarantee the safety of the Pareto optimal solutions obtained after solving the optimization problem in terms of WCET values.

Moreover, as mentioned before, we currently lack an energy model that considers a DMA controller. Therefore, we cannot calculate the extra costs in terms of energy consumption due to arbitration and CPU stalls during runtime. However, given an extended energy model for the DMA controller, a similar analyzer can be employed to analyze the extra cost in terms of energy consumption due to runtime DMA arbitration and CPU stalls.

Lastly, we use the WCET and energy analyzers to perform respective analyses of this temporary static version of the code (Line 12). Once the static version of the code is analyzed, we collect the results and discard this temporary code version (cf. Figure 9.3).

9.3 DMA Call Placement Optimization

Dynamic SPM allocation enables clever utilization of the memory hierarchy. But, using the CPU for dynamic copying engages it in non-essential work. As mentioned in Section 2.4.3, a DMA controller can work in parallel with the CPU and carry out dynamic copying, leaving the CPU free for essential execution. As the Flash and SPM considered in this chapter are single-ported memories, only one master can access a memory at any given time. Consequently, even if we configure multiple DMA channels for transfers, only one DMA channel can access the memory at once. Therefore, we only initiate a single memory-to-memory DMA transfer at a time to avoid unnecessary pessimism in our proposed DMA analysis model.

After solving the DCPO problem, we can perform DMA transfer and execute code using the CPU simultaneously. Whenever the CPU requires access to the bus matrix, arbitration takes place, and the CPU is granted access to the bus. For the DMA controller model considered in this chapter, we assume that if a second DMA transfer is triggered before the first one is complete, we stall the CPU to finish the first DMA transfer in burst mode. After that, the DMA controller initiates the second DMA transfer in arbitration mode.

This section is divided into the following subsections: Subsection 9.3.1 provides an overview of DMA call placement with examples. Subsection 9.3.2 proposes a model for the cost calculation of DMA transfer calls. Subsection 9.3.3 uses the cost calculation model and proposes the DCPO optimization problem that aims to minimize $\mathcal{T}'(y)$.

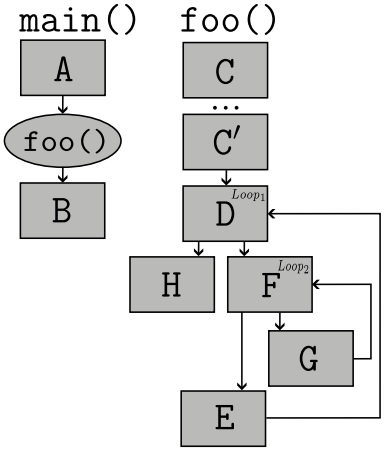
9.3.1 Contextual Overview

To dynamically allocate address objects from the Flash memory to SPM during runtime, we need to place calls for DMA transfer before executing the respective memory object. If the DMA transfer is incomplete before the execution reaches the respective memory object, the CPU is stalled, and the DMA transfer takes place in burst mode. Therefore, generally speaking, DCPO's strategy is to place DMA calls such that they completely or partially run in arbitration mode, and if it is not possible due to some constraint, the DMA transfer runs in burst mode.

Figure 9.4 provides a CFG of the exemplary program from Listing 8.1 in Chapter 8. In Figure 9.4, if we place DMA calls to dynamically allocate $\mathtt{Loop}_1$ to SPM between basic blocks `C'` and `D`, the DMA transfer of $\mathtt{Loop}_1$ needs to end before executing `D`. In this case, the CPU is stalled, and the DMA transfer occurs in the burst mode. Here, we do not benefit from the parallelization potential of the DMA controller.

To minimize the CPU stall time, we need to find appropriate places within the code so that the CPU and the DMA controller work in parallel. Therefore, we propose a single-objective DMA Call Placement Optimization (DCPO) that

Figure 9.4.: CFG of the exemplary program code from Listing 8.1



aims to find optimal places for DMA calls within the code so that the total execution time cost ($\mathcal{T}'(y)$) required for DMA transfers in arbitration mode during runtime is minimized.

As mentioned before, since we currently lack an energy model for a DMA controller, we do not consider energy consumption a key objective while solving the proposed DCPO optimization. However, if an energy model for a DMA controller is available, the method used to calculate the total execution time cost in arbitration mode (cf. Section 9.3.2) could be expanded and applied to model the energy consumption behavior of the DMA controller. Moreover, DCPO could be expanded to a multi-objective optimization similar to MO_{DMA} and solved using metaheuristic algorithms. However, these considerations are out of the scope of this thesis.

Figures 9.5, 9.6, and 9.7 provide examples with different probable placements for two DMA transfer calls for $Loop_1$ and $Loop_2$, where we assume that both $Loop_1$ and $Loop_2$ are dynamically allocated to SPM. As mentioned in Section 8.3.3, the address assignment algorithm might try to place address objects associated with $Loop_1$ and $Loop_2$ in consecutive SPM addresses if they are in consecutive Flash memory addresses. In such a scenario, we might need just one DMA transfer call to transfer both $Loop_1$ and $Loop_2$ together to SPM. However, for brevity's sake, in this example, we assume that two DMA calls are needed to transfer $Loop_1$ and $Loop_2$. In Figure 9.5, we assumed that there are a bunch of basic blocks indicated by "..." between `C` and `K`, `K` and `L`, and `L` and `C'`. Similarly, in Figures 9.6 and 9.7, "..." always indicates a bunch of basic blocks between respective surrounding basic blocks.

Figure 9.5 shows that the DMA transfer calls are placed after basic blocks `K` and `L`. In this case, the DMA transfers of $Loop_1$ and $Loop_2$ are initiated after the execution of basic blocks `K` and `L`, respectively. If the DMA transfer of $Loop_1$ is not complete before the DMA call for $Loop_2$, the CPU will be stalled after executing `L`, and the DMA transfer of $Loop_1$ will be carried out in burst mode before initializing the DMA transfer of $Loop_2$. However, if $Loop_1$ is

Figure 9.5.: Exemplary DMA call placements with arbitration

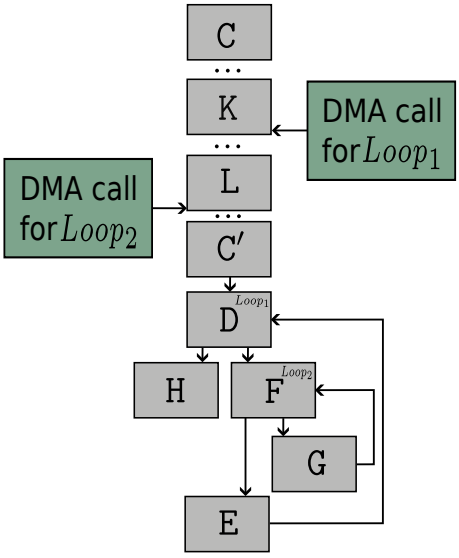
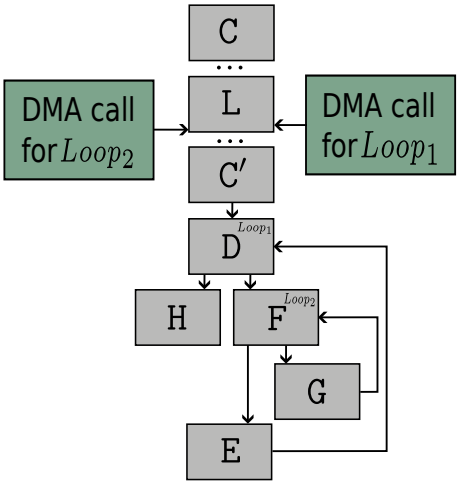


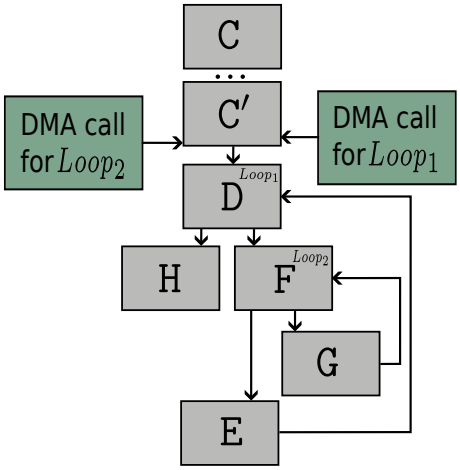
Figure 9.6.: Exemplary consecutive DMA call placements



transferred before initializing the DMA call for $Loop_2$, the DMA transfer of both $Loop_1$ and $Loop_2$ occurs with arbitration, and the CPU is granted bus access whenever needed.

As mentioned in Section 2.4.3, we can use the interrupt or polling mechanisms to ensure that the first DMA transfer is completed before initiating the second and that the DMA channel is free. For example, in the case of the interrupt mechanism, a `DMA_IT_TC` flag can be used to signal that the DMA transfer is done for the STM32F0xx microcontroller. In the case of the polling mechanism, a `DMA_FLAG_TC1` flag is used by the STM32F0xx microcontroller to poll the status of the DMA transfer while using DMA channel 1 [STM22a]. In this chapter, we use the polling mechanism during the evaluation to ensure that the first DMA transfer is completed before initiating the second. For example, in Figure 9.5, if the DMA transfer of $Loop_1$ is incomplete before initializing the

Figure 9.7.: Exemplary DMA call placements in burst mode



DMA call for $Loop_2$, the CPU is stalled until the polling flag (`DMA_FLAG_TC1`) indicates its completion.

Figure 9.6 shows that both DMA transfer calls are placed consecutively after basic block `L`. In this scenario, the DMA transfer of $Loop_1$ occurs in burst mode, and the CPU will be halted in the meantime. Then, in parallel, the DMA transfer of $Loop_2$ and the instruction execution by the CPU start. The CPU will be granted bus access whenever needed during the DMA transfer of $Loop_2$.

Figure 9.7 shows that both the DMA calls for $Loop_1$ and $Loop_2$ are placed after basic block `C'`. The DMA call for $Loop_1$ will occur in the burst mode due to two reasons. First, the DMA controller must transfer $Loop_1$ completely before initializing another DMA call. Second, to execute basic block `D` from SPM, the $Loop_1$ DMA transfer needs to be completed. In the case of the DMA call for $Loop_2$, basic block `D` belongs to the live range of the $Loop_2$ memory object—$Loop_2$ is nested within $Loop_1$. Therefore, the CPU will be stalled until the DMA transfer of $Loop_1$ and $Loop_2$ is complete.

Depending on the placement of the DMA calls, one of the three cases described above will be evoked. If $Loop_1$ and $Loop_2$ are smaller in size and DMA transfer in burst mode takes less time than in arbitration mode (arbitration for bus access costs extra cycles), then cases described in Figures 9.6 or 9.7 could be optimal placements. However, if $Loop_1$ and $Loop_2$ are larger in size and the extra cycles required for arbitration are less than the time taken by the burst mode DMA transfer, then the case described in Figure 9.5 could be the optimal DMA call placement.

If the DCPO optimization cannot find optimal places within the code to place DMA calls that fulfill all constraints, then the DMA calls will be placed before the *def* basic blocks of the memory objects, evoking the case described in Figure 9.7. Therefore, the proposed DCPO optimization considers all such cases and aims to find optimal places for all DMA transfer calls within the code to minimize the total execution time required for the DMA transfer in arbitration mode.

9.3.2 Cost Calculation

In this chapter, we built the DMA transfer model based on the DMA-related information presented in [STM22b] and [WA09]. We model the execution time cost of loading β bytes from the Flash to SPM using the DMA controller in burst mode as:

$$t_s(\beta) = \lceil \beta * D_{tr} \rceil + t_\Omega, \quad \text{where} \quad t_\Omega = D_{at} + aT_F + aT_S \tag{9.3}$$

D_{tr} is the DMA transfer rate; as the unit for t_s is cycles, the unit for D_{tr} is considered cycles/byte in this equation, i.e., the number of cycles required to transfer each byte. Within WCC, we use the maximum DMA transfer rate of the processor given in MB/s and the clock frequency of the processor given in MHz to calculate D_{tr} in cycles/byte. t_Ω is the total number of cycles required to perform arbitration once, D_{at} is the DMA arbitration time, and aT_F and aT_S are access times for the Flash and SPM. The bus matrix needs to grant access to the DMA controller to perform a DMA transfer in burst mode. Consequently, the arbitration cost term (t_Ω) is added to the cost of the burst mode operation. The system designer can freely set these parameters within the WCC compiler according to the processor board under consideration. We must insert one DMA call per address object allocated to the SPM. Therefore, the number of DMA calls within the code is always equal to the number of address object candidates for DMA transfer. $\delta_{(p,m,a)}$ is the DMA call for the a^{th} address object of the m^{th} memory object in the p^{th} function.

In the DMA cost calculation model proposed in this chapter, we do not explicitly consider the effects of a multi-stage pipeline in the underlying architecture. Consequently, this pipeline-unaware model can be pessimistic, as it does not account for potential parallelism between the CPU and the DMA controller arising from instruction pipelining. As mentioned previously, the pipeline-unaware model presented in this chapter has already been proposed in [JF25]. However, in Appendix F, we introduce a theoretical pipeline-aware DMA cost model for a 3-stage pipeline architecture. This pipeline-aware model is included only in the appendix and not in the main body of the thesis, as its practical implementation within the WCC compiler is currently missing. In future work, this pipeline-aware model could be implemented within the compiler and compared against the pipeline-unaware model proposed in this chapter.

This subsection is divided further into the following subsections: In Subsection 9.3.2.1, we first calculate the total execution time cost for DMA transfers in burst mode ($\mathcal{T}(x)$). In Subsection 9.3.2.2, we calculate the placement cost for each basic block, assuming that the DMA transfer and the execution of the basic block occur in parallel. In Subsection 9.3.2.3, we calculate the total execution time cost in the arbitration mode using the basic block-level placement cost calculated in Subsection 9.3.2.2.

9.3.2.1 Total Execution Time Cost in Burst Mode

As mentioned before, in burst mode, the complete DMA transfer occurs in one go without any CPU interruptions. Therefore, the total execution time cost required by the DMA controller to perform dynamic allocation during runtime only depends on the total number of address objects that need allocation and their sizes. Let $\mathcal{T}(x)$ be the total execution time cost for the DMA transfer in the burst mode for the solution vector x.

$$\mathcal{T}(x) = \sum_{p=1}^{P} \sum_{m=1}^{M_p} \left(\sum_{a=1}^{A_{(p,m)}} t_s(\beta_{(p,m,a)}) \right) x_{(p,m)} \tag{9.4}$$

where $\beta_{(p,m,a)}$ is the size of the address object $\mathcal{A}_{(p,m,a)}$, and $t_s(\beta_{(p,m,a)})$ is the cost of loading $\mathcal{A}_{(p,m,a)}$ from the Flash memory to SPM using the DMA controller during runtime.

9.3.2.2 Basic Block-level Placement Cost in Arbitration Mode

In contrast to the burst mode, the cost calculation in arbitration mode is more complex. In arbitration mode, the DMA and CPU aim to operate in parallel. However, suppose the CPU requires access to the bus matrix to fetch instructions from memory or to execute load or store operations. In that case, the CPU is granted access to the bus, causing the DMA transfer to halt temporarily. The total execution time cost required by the DMA controller in arbitration mode includes the times when the arbitration occurs, halting the DMA transfer. As mentioned before, we do not explicitly consider the effects of a multi-stage pipeline in this DMA cost calculation model. However, this model considers the CPU accesses to the bus matrix during instruction fetching and execution of load or store instructions. In this subsection, we will first calculate the placement cost (the extra execution time) for each basic block. Here, we assume that the DMA transfer and the execution of the basic block occur in parallel.

Let us first define some terms used to calculate the basic block-level placement cost. $r = \overline{1, B_p}$ is an index to indicate a basic block (b_p^r) where a DMA transfer call might potentially be placed, and B_p is the total number of basic blocks in the p^{th} function f_p. If a DMA transfer call $\delta_{(p,m,a)}$ is placed after the r^{th} basic block b_p^r, the placement cost for each basic block between b_p^r and the *def* basic block of the memory object $\mathcal{M}_{(p,m)}$ needs to be calculated to estimate the total execution time in the arbitration mode. The index j indicates the basic blocks between the r^{th} and the *def* basic block.

Let $\mathbf{N}_\Omega^j = \mathbf{N}^j + \mathbf{N}_l^j + \mathbf{N}_s^j$ be the maximum number of times arbitration can occur for the j^{th} basic block, i.e., the maximum number of times the CPU might need to access the memory while executing the j^{th} basic block. $\mathbf{N}^j$ is the total number of instructions, $\mathbf{N}_l^j$ is the total number of load instructions,

and $\mathbf{N}_s^j$ is the total number of store instructions in the j^{th} basic block. For each instruction, the CPU might need to access memory during the instruction fetching phase. For load and store instructions, the CPU might need to access memory not only during instruction fetching but also during the execution phase. Therefore, $\mathbf{N}_\Omega^j$ represents the worst-case scenario where the CPU might need bus access for every instruction.

To solve the DCPO optimization problem using the proposed DMA cost model, we need to perform a Best-Case Execution Time (BCET) analysis first. We perform BCET analysis to calculate BCET values for all basic blocks and use them to calculate the total execution time cost required for DMA transfers. We perform BCET analysis using a compiler-internal analyzer and calculate the BCET values for each basic block.

Let t_B^j be the BCET of the j^{th} basic block. BCET provides the minimum execution time of each basic block, i.e., the minimum time the DMA and the CPU can run in parallel. As we subtract this term from the DMA call placement cost and eventually, the total DMA call placement cost $\mathcal{T}'(y)$ is added to the WCET objective of the MO_{DMA} problem (cf. Equation 9.1), we use BCETs so that the DMA call placement cost does not violate the worst-case execution scenario. It implies that the total DMA call placement cost is an overestimation in average-case scenarios, but is valid for the worst-case scenario.

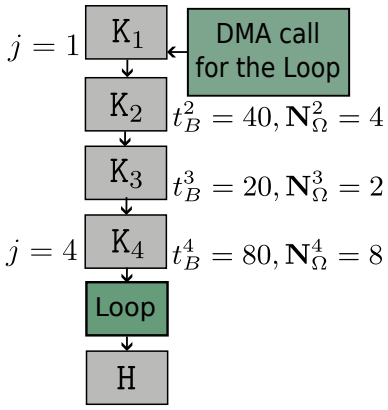
Let t_R^j represent the relative placement cost for the j^{th} basic block, valid for all basic blocks where $j > (r+1)$. As previously mentioned, the DMA transfer call $\delta_{(p,m,a)}$ for the address object $\mathcal{A}_{(p,m,a)}$ is placed after the r^{th} basic block of the function f_p.

$$t_R^j = \sum_{j'=r}^{(j-1)} t_{(p,m,a)}^{j'} - t_B^j \tag{9.5}$$

The term t_R^j is calculated by subtracting the BCET of the current j^{th} basic block (t_B^j) from the sum of placement costs of previous basic blocks ($\sum_{j'=r}^{(j-1)} t_{(p,m,a)}^{j'}$). Using t_R^j, we want to check if the DMA transfer of the address object $\mathcal{A}_{(p,m,a)}$ might end during the execution of the current j^{th} basic block. I.e., if $t_R^j \leqslant 0$, the DMA transfer completes during the execution of the current j^{th} basic block.

Let $t_{(p,m,a)}^j$ represent the placement cost incurred due to the j^{th} basic block between the r^{th} basic block and the *def* basic block of $\mathcal{A}_{(p,m,a)}$. Depending on the distance between the r^{th} and j^{th} basic block and the number of bytes transferred by the DMA controller, the cost value of $t_{(p,m,a)}^j$ is decided. The amount of DMA transfer done is decided using the relative placement cost t_R^j from Equation 9.5. Therefore, we will apply the following equation with five different cases to calculate the cost value $t_{(p,m,a)}^j$:

Figure 9.8.: Exemplary CFG with DMA transfer call for DMA placement cost calculation: We assume that $\forall j = \overline{1,4}$, $r = 1$, and arbitration time $t_\Omega = 2$ cycles.



$$t^j_{(p,m,a)} = \begin{cases} t_s(\beta_{(p,m,a)}), & \text{if } j = r, \\ (\mathbf{N}^j_\Omega + 1)t_\Omega - t_s(\beta_{(p,m,a)}), & \text{if } t^j_R \leqslant 0 \wedge j = r+1, \\ \mathbf{N}^j_\Omega t_\Omega - t^j_B, & \text{if } t^j_R > 0 \wedge j > r, \\ (\mathbf{N}^j_\Omega + 1)t_\Omega + & \text{if } t^{j-1}_R > 0 \wedge t^j_R \leqslant 0 \wedge j > r+1, \\ (\sum_{j'=(r+1)}^{(j-1)} t^{j'}_B - t_s(\beta_{(p,m,a)})), & \\ 0, & \text{if } t^{j-1}_R \leqslant 0 \end{cases} \tag{9.6}$$

where t_Ω is the total number of cycles required to perform arbitration once (cf. Equation 9.3). We will now examine each case outlined in Equation 9.6 and clarify them using examples in Figures 9.8, 9.9, 9.10, and 9.11. Figure 9.8 presents an exemplary CFG with one DMA transfer call that transfers a "Loop", and Figures 9.9, 9.10, and 9.11 provide three example scenarios with different DMA call costs in burst mode ($t_s(\beta_{(p,m,a)})$) to show the calculations for DMA call placement cost. In Figure 9.8, the DMA transfer call is placed between basic blocks K_1 and K_2, i.e., the DMA transfer call is placed after basic block $r = 1$. We assume the BCET values (t^j_B) and $\mathbf{N}^j_\Omega$ values for basic blocks K_2, K_3, and K_4 in Figure 9.8 to perform representative calculations shown in illustrations 9.9, 9.10, and 9.11. Depending on the value of j and t^j_R, the corresponding case in Equation 9.6 decides the cost incurred by the j^{th} basic block as follows:

Case 1 $t^j_{(p,m,a)} = t_s(\beta_{(p,m,a)})$, if $j = r$ **:**

As the DMA call $\delta_{(p,m,a)}$ is placed after the r^{th} basic block, $j = r$ is the first basic block where we start counting the cost for $t^j_{(p,m,a)}$. The placement cost for the $(j = r)^{th}$ basic block is equal to the cost of transferring the address object $\mathcal{A}_{(p,m,a)}$ of size $\beta_{(p,m,a)}$ bytes in burst

Figure 9.9.: Exemplary DMA placement cost calculation: We assume that $\forall j = \overline{1,4}$, $r = 1$, arbitration time $t_\Omega = 2$ cycles, and burst mode DMA call cost $t_s(\beta_{(p,m,a)}) = 40$ cycles

- $t^1_{(p,m,a)} = t_s(\beta_{(p,m,a)}) = 40$
- $t^2_R = 0 \implies t^2_{(p,m,a)} = (N^2_\Omega + 1) * t_\Omega - t_s(\beta_{(p,m,a)}) = -30$
- $t^3_R = -10$ and $t^2_R \leqslant 0 \implies t^3_{(p,m,a)} = 0$
- $t^4_R = -70$ and $t^3_R \leqslant 0 \implies t^4_{(p,m,a)} = 0$
- Total cost $= \sum_{j=1}^{4} t^j_{(p,m,a)} = 10$ cycles

mode ($t_s(\beta_{(p,m,a)})$). If any other DMA call ($\delta_{(p',m',a')}$) for some $\mathcal{A}_{(p',m',a')}$ is placed immediately after $\delta_{(p,m,a)}$, the DMA transfer for $\mathcal{A}_{(p,m,a)}$ is carried out in the burst mode before initiating the second DMA call $\delta_{(p',m',a')}$ for $\mathcal{A}_{(p',m',a')}$ (cf. Section 9.3.1 and Figure 9.6). In this case, the $j = r$ case accounts for the total cost of transferring $\mathcal{A}_{(p,m,a)}$ in the burst mode. However, if the DMA transfer of $\mathcal{A}_{(p,m,a)}$ is not interrupted by another DMA call, the DMA transfer occurs in arbitration mode in parallel with the execution of the succeeding basic blocks. I.e., the cost value of $t^j_{(p,m,a)}$ is calculated for $j > r$ cases using the other four cases in Equation 9.6. In Figures 9.9, 9.10, and 9.11, $t^1_{(p,m,a)}$ is always equal to the respective $t_s(\beta_{(p,m,a)})$ value.

Case 2 $t^j_{(p,m,a)} = (N^j_\Omega + 1)t_\Omega - t_s(\beta_{(p,m,a)})$, if $t^j_R \leqslant 0 \wedge j = r + 1$ **:**

The relative placement cost (t^j_R) indicates if the DMA transfer is complete during the execution of the current basic block ($t^j_R \leqslant 0$). If the DMA transfer of $\mathcal{A}_{(p,m,a)}$ is complete during the execution of the $(j = r + 1)^{th}$ basic block, then the total cost of transferring $\mathcal{A}_{(p,m,a)}$ should only include the arbitration costs that were incurred during the execution of the $(j = r + 1)^{th}$ basic block, i.e., $(N^j_\Omega + 1)t_\Omega$. We add a "+1" to compensate for the arbitration cost t_Ω from $t_s(\beta_{(p,m,a)})$ that is being subtracted.

In Figure 9.9, we assume the DMA call cost to transfer the Loop in burst mode is $t_s(\beta_{(p,m,a)}) = 40$ cycles. This example is structured so that the DMA transfer completes during the execution of the basic block K_2, even if K_2 executes as fast as possible under the best-case conditions. In this scenario, the relative cost t^2_R becomes zero, representing that the burst DMA costs are fully processed while K_2 executes. Consequently,

Figure 9.10.: Exemplary DMA placement cost calculation: We assume that $\forall j = \overline{1,4}$, $r = 1$, arbitration time $t_\Omega = 2$ cycles, and burst mode DMA call cost $t_s(\beta_{(p,m,a)}) = 120$ cycles

- $t^1_{(p,m,a)} = t_s(\beta_{(p,m,a)}) = 120$
- $t^2_R = 80 \implies t^2_{(p,m,a)} = \mathbf{N}^2_\Omega t_\Omega - t^2_B = -32$
- $t^3_R = 68 \implies t^3_{(p,m,a)} = \mathbf{N}^3_\Omega t_\Omega - t^3_B = -16$
- $t^4_R = -8 \implies t^4_{(p,m,a)} =$
  $(\mathbf{N}^4_\Omega + 1)t_\Omega + (\sum_{j'=2}^{3} t^{j'}_B - t_s(\beta_{(p,m,a)})) = -42$
- Total cost $= \sum_{j=1}^{4} t^j_{(p,m,a)} = 30$ cycles

Figure 9.11.: Exemplary DMA placement cost calculation: We assume that $\forall j = \overline{1,4}$, $r = 1$, arbitration time $t_\Omega = 2$ cycles, and burst mode DMA call cost $t_s(\beta_{(p,m,a)}) = 200$ cycles

- $t^1_{(p,m,a)} = t_s(\beta_{(p,m,a)}) = 200$
- $t^2_R = 160 \implies t^2_{(p,m,a)} = \mathbf{N}^2_\Omega t_\Omega - t^2_B = -32$
- $t^3_R = 148 \implies t^3_{(p,m,a)} = \mathbf{N}^3_\Omega t_\Omega - t^3_B = -16$
- $t^4_R = 152 \implies t^4_{(p,m,a)} = \mathbf{N}^4_\Omega t_\Omega - t^4_B = -64$
- Total cost $= \sum_{j=1}^{4} t^j_{(p,m,a)} = 88$ cycles

the only additional costs incurred are the arbitration costs during the execution of K_2.

From calculations, we can see that $(t^2_R = 0) \leqslant 0$ and $(j = 2) = (r + 1)$ as $r = 1$. Therefore, the second case in Equation 9.6 is already satisfied at the basic block K_2. Consequently, the total cost incurred to transfer the Loop is 10 cycles, equal to the arbitration cost $((\mathbf{N}^j_\Omega + 1)t_\Omega)$ incurred until the execution of K_2. As the DMA transfer of the Loop is over during the execution of K_2 itself, the value of t^3_R and t^4_R is set to zero using the fifth case in Equation 9.6, which is explained later in detail.

Case 3 $t^j_{(p,m,a)} = \mathbf{N}^j_\Omega t_\Omega - t^j_B$, if $t^j_R > 0 \wedge j > r$ **:**

If the DMA transfer of $\mathcal{A}_{(p,m,a)}$ is not completed ($t_R^j > 0$) during the execution of the j^{th} basic block, where $j > r$, then the third case in Equation 9.6 calculates the number of cycles during which the DMA and the CPU operate in parallel. Since we are subtracting the execution time of the basic block from the arbitration cost incurred during the execution of the j^{th} basic block, we use the BCET value of that basic block. This approach ensures that the $t^j_{(p,m,a)}$ cost is pessimistic and remains valid for the worst-case scenario.

In Figure 9.10, we assume the DMA cost to transfer the Loop in burst mode is $t_s(\beta_{(p,m,a)}) = 120$ cycles. From calculations, we can see that $(t_R^2 = 80) > 0$ and $(j = 2) > r$ as $r = 1$, and $(t_R^3 = 68) > 0$ and $(j = 3) > r$. Therefore, the third case in Equation 9.6 is satisfied for both t_R^2 and t_R^3. The negative values of $t^2_{(p,m,a)}$ and $t^3_{(p,m,a)}$ indicate that the DMA transfer of the Loop and the execution of basic blocks K_2 and K_3 are happening in parallel.

In Figure 9.11, we assume the cost to transfer the Loop in burst mode is $t_s(\beta_{(p,m,a)}) = 200$ cycles. Similar to the case in Figure 9.10, both t_R^2 and t_R^3 are greater than zero and satisfy the third case in Equation 9.6. However, we can also see from calculations that $(t_R^4 = 152) > 0$ and $(j = 4) > (r = 1)$, i.e., the third case in Equation 9.6 is also satisfied for the last basic block K_4 before the *def* basic block of the Loop. Consequently, the DMA transfer is not complete, and the CPU is stalled until the DMA transfer is complete before the execution of the Loop. Therefore, the total DMA transfer cost is 88 cycles, i.e., the number of cycles due to arbitration plus the extra time to complete the remaining DMA transfer in burst mode.

Case 4 $t^j_{(p,m,a)} = (\mathbf{N}^j_\Omega + 1)t_\Omega + (\sum_{j'=(r+1)}^{(j-1)} t_B^{j'} - t_s(\beta_{(p,m,a)}))$, if $t_R^{j-1} > 0 \wedge t_R^j \leqslant 0 \wedge j > r+1$ **:**

The fourth case in Equation 9.6 is satisfied if the DMA transfer of $\mathcal{A}_{(p,m,a)}$ is complete during the execution of any basic block where $j > r+1$. The condition $t_R^{j-1} > 0$ indicates that the DMA transfer was not completed during the execution of the $(j-1)^{th}$ basic block, while the condition $t_R^j \leqslant 0$ confirms that the DMA transfer is now complete. When the fourth case in Equation 9.6 is satisfied, we add the arbitration costs $((\mathbf{N}^j_\Omega + 1)t_\Omega)$ incurred during the execution of this basic block. The "+1" compensates for the arbitration cost t_Ω from $t_s(\beta_{(p,m,a)})$ that is being subtracted. We subtract $t_s(\beta_{(p,m,a)})$ to compensate for the burst mode $t_s(\beta_{(p,m,a)})$ cost that was added during the first $j = r$ case. Additionally, the $\sum_{j'=(r+1)}^{(j-1)} t_B^{j'}$ term is added to compensate for all the BCET values that were being subtracted from all previous j'^{th} basic blocks. Consequently,

if the DMA transfer is not interrupted by another DMA transfer call or if it is completed before the execution of $\mathcal{A}_{(p,m,a)}$ (as illustrated in Figure 9.5), the total number of cycles required for the DMA transfer in arbitration mode will include all possible arbitration costs.

In Figure 9.10, we assume the DMA call cost to transfer the Loop in burst mode is $t_s(\beta_{(p,m,a)}) = 120$ cycles. From calculations, we can see that $(t_R^4 = -8) \leqslant 0$ and $(j = 4) > (r + 1)$ as $r = 1$. Therefore, the fourth case in Equation 9.6 is satisfied. The loop transfer will finish during the execution of K_4, and the total DMA transfer cost is 30 cycles, i.e., the number of cycles due to arbitration from the initialization of the DMA call until the basic block K_4.

Case 5 $t^j_{(p,m,a)} = 0$, if $t_R^{j-1} \leqslant 0$ **:**

Lastly, if the DMA transfer of $\mathcal{A}_{(p,m,a)}$ is complete before the execution of the j^{th} basic block, i.e., $t_R^{j-1} \leqslant 0$, then the $t^j_{(p,m,a)}$ cost for all the remaining basic blocks is set to zero.

In Figure 9.9, the DMA transfer of the Loop is over during the execution of K_2 itself, and the DMA costs for K_3 and K_4 are set to zero, i.e., $t^3_{(p,m,a)} = t^4_{(p,m,a)} = 0$, as this fifth case in Equation 9.6 is satisfied, i.e., for $j = 3$, $t_R^{j-1} = t_R^2 = -30 \leqslant 0$ and for $j = 4$, $t_R^{j-1} = t_R^3 = -10 \leqslant 0$.

Using the examples in Figures 9.8, 9.9, 9.10, and 9.11, we can see possible scenarios where each case in Equation 9.6 is invoked. In Equation 9.6, we do not differentiate between if-else branches, i.e., in the case of an if-else statement between the r^{th} and the j^{th} basic block, all basic blocks are considered while modeling $t^j_{(p,m,a)}$ cost. We could make the $t^j_{(p,m,a)}$ cost model more robust in the future by considering such scenarios. In the next subsection, we use Equation 9.6 to calculate the total execution time in arbitration mode to perform DMA transfers during runtime.

9.3.2.3 Total Execution Time Cost in Arbitration Mode

Let us first define some terms that will be used to calculate the total execution time cost in arbitration mode ($\mathcal{T}'(y)$). $y^r_{(p,m,a)} \in \{0, 1\}\ \forall r = \overline{1, (B_p - 1)}$ is the binary decision variable vector that decides whether the DMA transfer call $\delta_{(p,m,a)}$ will be placed after the r^{th} basic block (b^r_p) within the p^{th} function or not, i.e.,

$$y^r_{(p,m,a)} = \begin{cases} 1, & \text{if } x_{(p,m)} = 1 \wedge \delta_{(p,m,a)} \text{ is placed after } b^r_p, \\ 0, & \text{otherwise} \end{cases} \tag{9.7}$$

As previously mentioned, B_p is the total number of basic blocks in function f_p. As we do not place a DMA call after the function exit basic block of any

function, the size of the binary decision variable vector $y_{(p,m,a)}$ for $\delta_{(p,m,a)}$ is $(B_p - 1)$. Moreover, let $\nu_{(p,m)} = (\nu^1_{(p,m)}, \ldots, \nu^{(B_p-1)}_{(p,m)})$ represent a binary vector for the memory object $\mathcal{M}_{(p,m)}$, where

$$\nu^q_{(p,m)} = \begin{cases} 1, & \text{if } b^q_p \text{ is the } \textit{def} \text{ basic block of } \mathcal{M}_{(p,m)}, \\ 0, & \text{otherwise} \end{cases} \tag{9.8}$$

As *def* basic blocks of memory objects are always the same, $\nu_{(p,m)}$ is a constant binary vector in the DCPO optimization. Additionally, $\nu^q_{(p,m)} = 1$ implies that the q^{th} basic block is the *def* basic block of $\mathcal{M}_{(p,m)}$.

$$\eta^j_{(p,m,a)} = \begin{cases} 1, & \text{if } j = r \vee (\sum_{m=1}^{M_p} \sum_{a'=1, a' \neq a}^{A_{(p,m)}} y^j_{(p,m,a')} = 0 \wedge \\ & \qquad \eta^k_{(p,m,a')} = 1 \; \forall k \in j : k < j \;), \\ 0, & \text{otherwise} \end{cases} \tag{9.9}$$

Next, $\eta^j_{(p,m,a)}$ is a binary variable that indicates if the basic block-level placement cost $t^j_{(p,m,a)}$ (cf. Equation 9.6) for the j^{th} basic block will be considered or not while calculating the total execution time cost. As we can see from Equation 9.9, $\eta^j_{(p,m,a)}$ is always "1" for the r^{th} basic block, i.e., $j = r$. Even if any other DMA call $\delta_{(p,m,a')}$ is placed immediately after the current DMA call $\delta_{(p,m,a)}$, $\eta^j_{(p,m,a)} = 1$ for $j = r$ ensures that at least the cost to transfer DMA call $\delta_{(p,m,a)}$ in burst mode is incurred. This case works similarly to the first case in Equation 9.6. Moreover, Figure 9.6 addresses this scenario with an example where the DMA call for $Loop_2$ is placed immediately after the DMA call for $Loop_1$. In this case, $\eta^j_{(p,m,a)}$ will be equal to "1" only for $j = r$, and $Loop_1$ DMA transfer will occur in burst mode, and then, DMA call for $Loop_2$ takes place in arbitration mode.

For all other basic blocks, $\eta^j_{(p,m,a)}$ is "1" only if no other DMA transfer calls are placed until $(j = (q-1))^{th}$ basic block. As previously mentioned, q^{th} basic block is the *def* basic block of the memory object $\mathcal{M}_{(p,m)}$, and all the address objects $\mathcal{A}_{(p,m,a)}$ are associated with $\mathcal{M}_{(p,m)}$. The $a' \neq a$ condition from Equation 9.9 implies that $\mathcal{A}_{(p,m,a)}$ and $\mathcal{A}_{(p,m,a')}$ are two different address objects; however, they can belong to the same memory object. If another DMA call $\delta_{(p,m,a')}$ is placed at the k^{th} basic block that is before the $(q-1)^{th}$ basic block, then $\eta^j_{(p,m,a)}$ is set to "0" from $j = (k+1)$ till $j = (q-1)$. Figure 9.5 addresses this scenario with an example, where $\eta^j_{(p,m,a)}$, for $Loop_1$, will be set to 0 for all the basic blocks after basic block L, as a second DMA transfer call for

$Loop_2$ is initiated after L. However, for $Loop_2$, as no other DMA call is placed between L and D, the value of $\eta^j_{(p,m,a)}$ is "1" for $Loop_2$ for all basic blocks.

Moreover, in the example in Figure 9.8, we assumed $\forall j = \overline{1,4}$, $\eta^j_{(p,m,a)} = 1$. However, if, for example, another DMA call for some other address object is placed after K_3 in Figure 9.8, then $\eta^j_{(p,m,a)}$ becomes zero $\forall j > 3$ (cf. Equation 9.9).

In Subsection 9.3.2.2, we calculated the basic block-level placement cost ($t^j_{(p,m,a)}$) for each j^{th} basic block in arbitration mode between the r^{th} and the q^{th} basic block. The DMA call $\delta_{(p,m,a)}$ is placed after the r^{th} basic block, and the q^{th} basic block is the *def* basic block of $\mathcal{M}_{(p,m)}$ and corresponding $\mathcal{A}_{(p,m,a)}$. Using $t^j_{(p,m,a)}$, we can determine the DMA call placement cost in arbitration mode incurred when $\delta_{(p,m,a)}$ is placed at the r^{th} basic block in the p^{th} function. Let $\sigma^r_{(p,m,a)}$ represent the DMA call placement cost in arbitration mode.

$$\sigma^r_{(p,m,a)} = \begin{cases} \sum_{j=r}^{(q-1)} t^j_{(p,m,a)} \eta^j_{(p,m,a)}, & \text{if } \nu^q_{(p,m)} = 1 \ \wedge \ r < (q-1) \\ t_s(\beta_{(p,m,a)}), & \text{if } \nu^q_{(p,m)} = 1 \ \wedge \ r = (q-1) \\ 0, & \text{otherwise} \end{cases} \tag{9.10}$$

The first case in Equation 9.10 captures the following cases: the DMA transfer with the arbitration (Figure 9.5) and if other DMA calls are placed after the current DMA call $\delta_{(p,m,a)}$ (Figure 9.6). The second case in Equation 9.10 captures cases like Figure 9.7, where the DMA call is placed immediately before the memory object, and the DMA transfer occurs in the burst mode.

Case 1 $\sigma^r_{(p,m,a)} = \sum_{j=r}^{(q-1)} t^j_{(p,m,a)} \eta^j_{(p,m,a)}$, if $\nu^q_{(p,m)} = 1 \ \wedge \ r < (q-1)$ **:**

As previously mentioned, $\nu^q_{(p,m)} = 1$ implies that the q^{th} basic block is the *def* basic block. The DMA call is placed after the r^{th} basic block such that at least one basic block exists between the r^{th} basic block and the *def* basic block of the memory object, i.e., $r < (q-1)$ (cf. Figures 9.5 and 9.6). In this case, we use the basic block-level placement cost ($t^j_{(p,m,a)}$) from Equation 9.6 to calculate the DMA call placement cost $\sigma^r_{(p,m,a)}$ incurred by $\delta_{(p,m,a)}$. When the DMA call is placed at $(j = r)^{th}$ basic block such that $r < (q-1)$, we perform a summation of $t^j_{(p,m,a)}$ incurred due to each basic block from $j = r$ until $j = (q-1)$. Depending on the value of $\eta^j_{(p,m,a)}$, the $t^j_{(p,m,a)}$ cost is either added to $\sigma^r_{(p,m,a)}$ or not. Once the value of $\eta^j_{(p,m,a)}$ becomes zero, $\eta^j_{(p,m,a)}$ is zero for the rest of the basic blocks.

Case 2 $\sigma^r_{(p,m,a)} = t_s(\beta_{(p,m,a)})$, if $\nu^q_{(p,m)} = 1 \ \wedge \ r = (q-1)$ **:**

If the DMA call $\delta_{(p,m,a)}$ is placed immediately before the *def* basic block of memory object $\mathcal{M}_{(p,m)}$, i.e., $r = (q-1)$, the CPU is stalled.

Then, the DMA controller transfers the address object $\mathcal{A}_{(p,m,a)}$ of the corresponding $\mathcal{M}_{(p,m)}$ in burst mode—the resulting cost is $t_s(\beta_{(p,m,a)})$. The memory object is executed from the SPM once the DMA transfer is complete.

Case 3 $\sigma^r_{(p,m,a)} = 0$, otherwise **:**

Since the DMA transfer must be completed before the execution of the memory object can begin, the value of the cost $\sigma^r_{(p,m,a)}$ is set to zero for all the basic blocks where $r > (q-1)$.

Finally, let $\mathcal{T}'(y)$ represent the total execution time cost required for all the DMA transfers in arbitration mode. Moreover, $\mathbb{F}^r_p$ is the execution frequency of the basic block b^r_p, where the DMA transfer call will be placed by the DCPO optimization.

$$\mathcal{T}'(y) = \sum_{p=1}^{P} \sum_{m=1}^{M_p} \left(\sum_{a=1}^{A_{(p,m)}} \sum_{r=1}^{B_p-1} \mathbb{F}^r_p \sigma^r_{(p,m,a)} y^r_{(p,m,a)} \right) x_{(p,m)} \tag{9.11}$$

The binary decision variable vector $y_{(p,m,a)}$ decides exactly where the DMA transfer call $\delta_{(p,m,a)}$ will be placed within the function. If $\delta_{(p,m,a)}$ is placed at the r^{th} basic block within function f_p, then the cost $\sigma^r_{(p,m,a)}$ calculated for the r^{th} basic block is added to the total $\mathcal{T}'(y)$.

$x_{(p,m)}$ is the decision vector that decides whether a memory object should be dynamically allocated to SPM or not. As the DCPO optimization problem is solved after performing address assignment, the decision vector $x_{(p,m)}$ is already set for the particular instance of DCPO evaluation (cf. Figure 9.3). Therefore, the decision vector $x_{(p,m)}$ is treated as a constant vector in Equation 9.11. In the next subsection, we formulate the DCPO optimization problem that aims to minimize the $\mathcal{T}'(y)$ objective value.

9.3.3 Optimization Problem

We want to find optimal places within the program to place the DMA calls such that the total execution time cost $\mathcal{T}'(y)$ required by the DMA controller in arbitration mode is minimized. We consider additional constraints while solving the DCPO problem:

1. DMA calls must always be placed before the corresponding memory object that is being executed. For example, if a function f_1 is called from function f_2 and f_1 needs to be dynamically allocated in arbitration mode to SPM during runtime, the DMA call that triggers the transfer of f_1 from f_2 should always be placed before the function f_1 is called for the first time from f_2. As we do not place the DMA calls for the corresponding memory objects after its *def* basic block, we can constrain those basic

blocks. $\nu^{q}_{(p,m)} = 1$ implies that the q^{th} basic block is the *def* basic block of the address object $\mathcal{A}_{(p,m,a)}$.

$$y^{r}_{(p,m,a)} = 0 \quad \forall r \geqslant q \mid \nu^{q}_{(p,m)} = 1 \tag{9.12}$$

2. DMA calls are not placed after basic blocks that are part of loops. If we place a DMA call inside a loop, the DMA transfer of the same address object will be triggered during every iteration. `isPartOfLoop`(b^{r}_{p}) is a Boolean function that checks if the basic block b^{r}_{p} is part of a loop or not.

$$y^{r}_{(p,m,a)} = 0 \quad \forall r \mid \texttt{isPartOfLoop}(b^{r}_{p}) = \textit{true} \tag{9.13}$$

3. DMA calls are placed only after the basic blocks that dominate the memory objects. If we place a DMA call within a branch of an if-else statement, which, when executed, never leads to the execution of the memory object, then triggering the DMA transfer within such a branch of the if-else statement is useless. Therefore, if the memory object is not dominated by the basic block, the DMA call will not be placed after it. b^{q}_{p} is the *def* basic block of the memory object, i.e., $\nu^{q}_{(p,m)} = 1$.

$$y^{r}_{(p,m,a)} = 0 \; \forall r \mid \nu^{q}_{(p,m)} = 1 \wedge \neg(b^{r}_{p} \text{ dom } b^{q}_{p}) \tag{9.14}$$

4. If two address objects belonging to different memory objects—$\mathcal{A}_{(p,m,a)}$ and $\mathcal{A}_{(p',m',a')}$, where $m \neq m'$—share an address range within SPM, and if $\mathcal{A}_{(p,m,a)}$'s *def* basic block is executed after $\mathcal{A}_{(p',m',a')}$, we must place the DMA call for $\mathcal{A}_{(p,m,a)}$ after $\mathcal{A}_{(p',m',a')}$'s execution to avoid overwriting $\mathcal{A}_{(p',m',a')}$ from SPM before its execution is finished. q_{last} is the index for the last basic block of the address object $\mathcal{A}_{(p',m',a')}$, and q_1 and q_2 are the indices for the *def* basic blocks for $\mathcal{A}_{(p,m,a)}$ and $\mathcal{A}_{(p',m',a')}$, respectively.

$$\begin{aligned} y^{r}_{(p,m,a)} = 0 \; \forall \, r \leqslant q_{last} \mid m \neq m' \wedge \\ (\, q_1 > q_2 \mid \nu^{q_1}_{(p,m)} = 1 \; \wedge \; \nu^{q_2}_{(p',m')} = 1 \,) \wedge \\ \gamma_{(p',m',a')} \leqslant \gamma_{(p,m,a)} + \beta_{(p,m,a)} \wedge \\ \gamma_{(p,m,a)} \leqslant \gamma_{(p',m',a')} + \beta_{(p',m',a')} \end{aligned} \tag{9.15}$$

5. Exactly one DMA call is placed for each address object.

$$\sum_{r=1}^{B_p} y^{r}_{(p,m,a)} = 1 \tag{9.16}$$

6. The total execution time cost with arbitration always remains less than or equal to the total execution time cost in burst mode.

$$\mathcal{T}'(y) \leqslant \mathcal{T}(x) \tag{9.17}$$

The final minimization problem for the DMA call placement optimization is mathematically formulated as follows:

$$\begin{aligned} \min_{y} \quad & \mathcal{T}'(y) \\ \text{subject to} \quad & \text{Eq. 9.12, Eq. 9.13, Eq. 9.14,} \\ & \text{Eq. 9.15, Eq. 9.16, and Eq. 9.17} \end{aligned} \tag{9.18}$$

DCPO is performed after the first address assignment stage in Algorithm 9.1. The address assignment provides DCPO with address objects, their source addresses in the Flash, their destination addresses in SPM, and their sizes. After solving the address assignment problem, we perform BCET analysis to get the BCET values for all basic blocks. This information serves as an input for the DCPO optimization.

We solve the DCPO optimization problem using three methods: FPA, SPEA, and ILP. As DCPO is a single-objective optimization, both metaheuristic- and ILP-based approaches can be used to solve it without tampering with the objective function.

To solve DCPO using FPA and SPEA, we generate an initial population randomly that constitutes individuals representing the binary decision vector y. During the metaheuristic algorithm run, this initial population is updated iteratively, checked if the updated individual fulfills all the constraints described in this section, and the individuals with better objectives are selected for the next iteration. If the individual does not satisfy the constraints described in this section, we try to repair the individual by flipping the corresponding bits and checking the constraint condition. In the worst case, DCPO will place all the DMA calls before the *def* basic blocks of the memory objects (cf. Figure 9.7), where the total execution time cost for DMA transfer is maximum, but all the constraint conditions are satisfied.

To solve DCPO using ILP, we define y as a binary ILP variable vector and apply all the constraints discussed in this section to build an ILP model. All the constraints and their conditions for DCPO are linear in nature—the solution vector x is treated as a constant vector while solving the DCPO problem. Therefore, they can be implemented easily as an ILP model within the WCC compiler. We solve the ILPs by feeding the ILP model as input to the ILP solver `Gurobi` [Gur24] within the WCC compiler.

By solving DCPO, we want to find the optimal decision vector y that satisfies all constraints described in this section and takes the minimum extra time to perform the required DMA transfers. During evaluations, we compare the

Algorithm 9.2 DMA-aware Multi-Objective Dynamic SPM Allocation (MO_{DMA})

```
1: Recognize and collect memory objects
2: Perform Liveness Analysis using Algorithm 8.1
3: for p = 1 : P do        ▷ Run metaheuristic algorithm for P different initial populations
4:     Randomly initialize the initial population of size N
5:     for n = 1 : N do
6:         Call Algorithm 9.1
7:     while Stopping criteria are not reached do
8:         Update Individuals using respective update operators
9:         for Each updated Individual do
10:            Call Algorithm 9.1
11:        Update the next generation using the selection operator
12:    Collect the Pareto optimal solution set
13: return Final Pareto optimal solution set (PF_F)
```

quality of the obtained solutions to determine the best approach—ILP, FPA, or SPEA—to solve the DCPO problem.

9.4 DMA-aware Multi-Objective Dynamic SPM Allocation

In this section, we propose the DMA-aware multi-objective dynamic SPM allocation (MO_{DMA}) problem that minimizes the program's WCET and energy consumption. Figure 9.3 provides an overview of all the steps involved in performing MO_{DMA}. In Section 9.2.1, we discussed and provided the mathematical formulation of the MO_{DMA} optimization problem. The MO_{DMA} problem is solved using a metaheuristic algorithm (FPA or SPEA), and we obtain final solutions as a set of Pareto optimal solutions.

Algorithm 9.2 presents the compiler-level MO_{DMA} optimization. To initialize the algorithm, we recognize and collect all the memory objects by performing standard control flow and depth-first analyses within WCC (Line 1). Then, we perform the liveness analysis to determine the live ranges of all memory objects (Line 2). We use the metaheuristic algorithms (FPA and SPEA) to solve the MO_{DMA} problem. We perform evaluations using P different sets of initial populations and run the metaheuristic algorithm to solve the MO_{DMA} problem P times for different initial populations (Line 3). We initialize the initial population of metaheuristic algorithms randomly (Line 4).

As the obtained Pareto front is an approximation, the initial population influences the convergence of the metaheuristic algorithm. Therefore, we generate the final Pareto front using the Pareto fronts obtained from each of

the P metaheuristic algorithm runs to reduce the influence of randomness on the final results (Lines 12 and 13).

Algorithm 9.1 is called to generate the dynamically allocated code for all the individuals in the initial population (Lines 5 and 6). After Algorithm 9.1 has performed address assignment, it solves the DCPO problem using either FPA, SPEA, or ILP, and the optimal objective value for $\mathcal{T}'(y)$ for the respective individual (decision vector $x \in X$) is obtained. DMA calls are placed within the program according to the placement positions provided by the solution vector y. Finally, a temporary static version of the dynamically allocated code is generated, the individual is analyzed using WCET and energy analyzers, and the objective values $F_1(x)$ and $F_2(x)$ are collected for each individual.

The maximum number of generations (*maxGen*) is one of the stopping criteria, as we want the metaheuristic algorithm to terminate at some point (Line 7). We specified the maximum number of generations for which the population remains the same as the second stopping criterion for the metaheuristic algorithm. The metaheuristic algorithm updates the individuals at every generation (Line 8). FPA uses local and global pollination operators, and SPEA uses recombination and mutation operators to update each individual at every iteration to explore the search space (cf. Section 4.4). Algorithm 9.1 is called to generate and analyze the dynamically allocated code for updated individuals, and their objective values are collected (Lines 9 and 10).

Based on the objective values, the metaheuristic algorithms use a selection operator based on Pareto dominance to collect the top-scoring solutions [ED18] and pass them to the next iteration (Line 11). After one of the two stopping criteria is met, the algorithm outputs the Pareto front (Line 12). We collect the Pareto fronts for each initial population and generate the final Pareto optimal solution set using Pareto dominance (Line 13).

9.5 Evaluation

In this section, we compare the proposed MO_{DMA} approach with the MO_D and MO_S optimization approaches from Chapters 8 and 5, respectively. Moreover, we compare the results from MO_{DMA} with single-objective SO_D [VM07] and SO_{DMA} [KBS17] approaches. We implemented versions of SO_{DMA} and SO_D within WCC. We also compare MO_{DMA} with the baseline BL evaluation, where the whole program is executed from the Flash. We solve all multi-objective optimizations using two metaheuristic algorithms—FPA and SPEA—and all single-objective optimizations are solved using ILPs. Furthermore, the DCPO optimization problem is solved using all three approaches—FPA, SPEA, and ILPs. Subsection 9.5.1 describes the overall evaluation setup used during these evaluations. During this evaluation, we aim to answer the following questions:

- Which is the best and the worst method among ILP, FPA, and SPEA to solve the DCPO problem?

- Even when considering the worst-performing DCPO approach, how does MO_{DMA} compare to other approaches for a fixed SPM size?
- How does MO_{DMA} compare to other approaches when considering the best-performing DCPO?
- Finally, what is the impact of different SPM sizes on the quality of solutions provided by the various approaches?

To answer these questions, the subsequent evaluation subsections are subdivided as follows:

- In Subsection 9.5.2, we solve the single-objective DCPO problem proposed in this chapter using FPA, SPEA, and ILP methods and evaluate the quality of solutions obtained using each method.
- In Subsections 9.5.3 and 9.5.4, we individually adjust ISPM and DSPM sizes to 60% relative to the benchmark size to pressurize the optimizations to allocate at most 60% of the benchmark. The size of the SPM plays a critical role in evaluating the performance of any dynamic or static SPM allocation-based optimization. If we consider a small-sized SPM compared to the benchmark size, no allocation might occur, resulting in no meaningful trends. On the other hand, if the SPM size is very big, the optimization might allocate the whole benchmark to SPM, again resulting in no meaningful trends. In Subsection 9.5.3, we solve the proposed MO_{DMA} optimization using DCPO–FPA, and in Subsection 9.5.4, we use DCPO–ILP to evaluate MO_{DMA}.
- Finally, in Subsection 9.5.5, we present the optimization results for 20%, 40%, and 80% relative ISPM and DSPM sizes, where only DCPO–FPA is used while evaluating MO_{DMA}. In this subsection, we aim to evaluate the impact of different SPM sizes on the quality of solutions.

9.5.1 Evaluation Setup

The evaluations are conducted on the ARM Cortex-M0 microcontroller's fork of the WCC compiler. We use the ILP solver `Gurobi` [Gur24] to solve the ILP-based optimization problems. WCET and energy analyses are conducted using `aiT` [Abs24] and `EnergyAnalyzer` [Tea20], respectively. The ARM Cortex-M0's energy model used within `EnergyAnalyzer` is discussed in Section 2.2. Wegener et al. [WNNYE23] and Nikov et al. [NGC+22] proposed an energy model with its associated costs that are used within the `EnergyAnalyzer` during energy analysis. The values for e_F and e_S in Equation 2.4 are taken from this energy model, i.e., $e_F = 1.037625441$ nJ and $e_S = 1.031341343$ nJ. The ARM Cortex-M0's clock frequency is set to 48 MHz. The memory layout used is

according to the STM32F0xx's layout recommendations [STM22a]. The access latencies considered are 6 cycles for the Flash (aT_F) and 1 cycle for SPM (aT_S). Additionally, the DMA transfer rate (D_{tr}) is set to 6.67 cycles/byte, and the DMA arbitration time (D_{at}) is set to 2 cycles [STM22b]. The system designer can freely configure parameters like clock frequency, aT_F, aT_S, D_{tr}, and D_{at} according to specific hardware specifications. The WCC compiler makes these parameters available to the system designer through a text-based interface.

Similar to Chapter 8, we evaluated 20 benchmarks from `AutoBench`, `Networking`, and `TeleBench` benchmark suites offered by the `Embedded Microprocessor Benchmark Consortium (EEMBC)` [PCLGO09]. We assume that the benchmarks will fit within the Flash memory. The code is generated by the compiler using the `-O2` optimization flag.

To generate DMA-aware code and trigger a DMA transfer, we use the Hardware Abstraction Layer (HAL) code for STM32F0xx. All the HAL code for STM32F0xx is compiled within the WCC compiler and linked to the compiled benchmark by the Linker. During startup, the DMA controller is initialized, and the DMA is configured for memory-to-memory transfer. During these evaluations, we selected DMA channel 1 (`DMA1_Channel1`) to perform DMA transfers during runtime. To start a DMA transfer, we call `HAL_DMA_Start`, which takes the following inputs: the source address (α), the destination address (γ), and the size of the address object (β). Moreover, `HAL_DMA_PollForTransfer` is used to check if the DMA transfer is complete before executing the corresponding code object or initiating the next DMA transfer. As mentioned in Section 9.2.2, these HAL-related functions are considered external during the WCET and energy analyses, and their contributions are therefore assumed to be zero.

For both FPA and SPEA algorithms, the values of hyperparameters are depicted in Tables 5.1 and 5.2, respectively. The number of initial populations for which the metaheuristic algorithm will run is set to 5, i.e., $P = 5$ (cf. Algorithm 9.2, Line 3). We need no extra parameter setting for ILP-based optimizations. The results obtained during these evaluations are valid for the algorithm parameters described above.

9.5.2 DCPO Evaluation

In this subsection, we compare the quality of FPA, SPEA, and ILP methods for solving the DCPO optimization problem proposed in Section 9.3. For this purpose, we run the MO_{DMA} optimization to generate dynamically allocated individuals. The MO_{DMA} optimization is performed in this subsection just to get the dynamically allocated individuals (x) that serve as inputs for the DCPO optimization problem. To find the best placement for DMA calls for each individual, we solve the DCPO problem using FPA, SPEA, and ILP, and compare the quality of the obtained solutions by comparing DCPO's minimized objective value ($\mathcal{T}'(y)$). Let $\mathcal{T}'_F$, $\mathcal{T}'_S$, and $\mathcal{T}'_I$ be the total DMA execution time for

the DMA transfer in arbitration mode (cf. Equation 9.11) for each individual obtained using FPA, SPEA, and ILP methods, respectively.

Generally, while solving the MO_{DMA} problem, the DCPO problem is solved only once using either FPA, SPEA, or ILP. However, as we are solving the DCPO problem three times for the same solution vector x in this subsection, we need to select one of the objective values ($\mathcal{T}'(y)$), which we obtain after solving the DCPO problem three times. As we do not strive to get the best possible Pareto solutions using MO_{DMA} in this section, we can just pick any one $\mathcal{T}'(y)$ value to eventually add to the WCET objective (cf. Equation 9.1). Therefore, we chose to select the worst objective value $\mathcal{T}'(y)$. This $\mathcal{T}'(y)$ is added to the WCET value of the solution vector x to advance the population generations in Algorithm 9.2. We insert DMA calls according to the worst DCPO solution, compile dynamically allocated code, and continue running Algorithms 9.1 and 9.2 to generate more individuals for evaluation purposes.

For comparison and finding the best approach to solve the DCPO problem, we use a standard normalization or scaling transformation on $\mathcal{T}'_F$, $\mathcal{T}'_S$, and $\mathcal{T}'_I$ to scale their values between 0 and 100. Therefore, we define the following relative execution metric (T_κ):

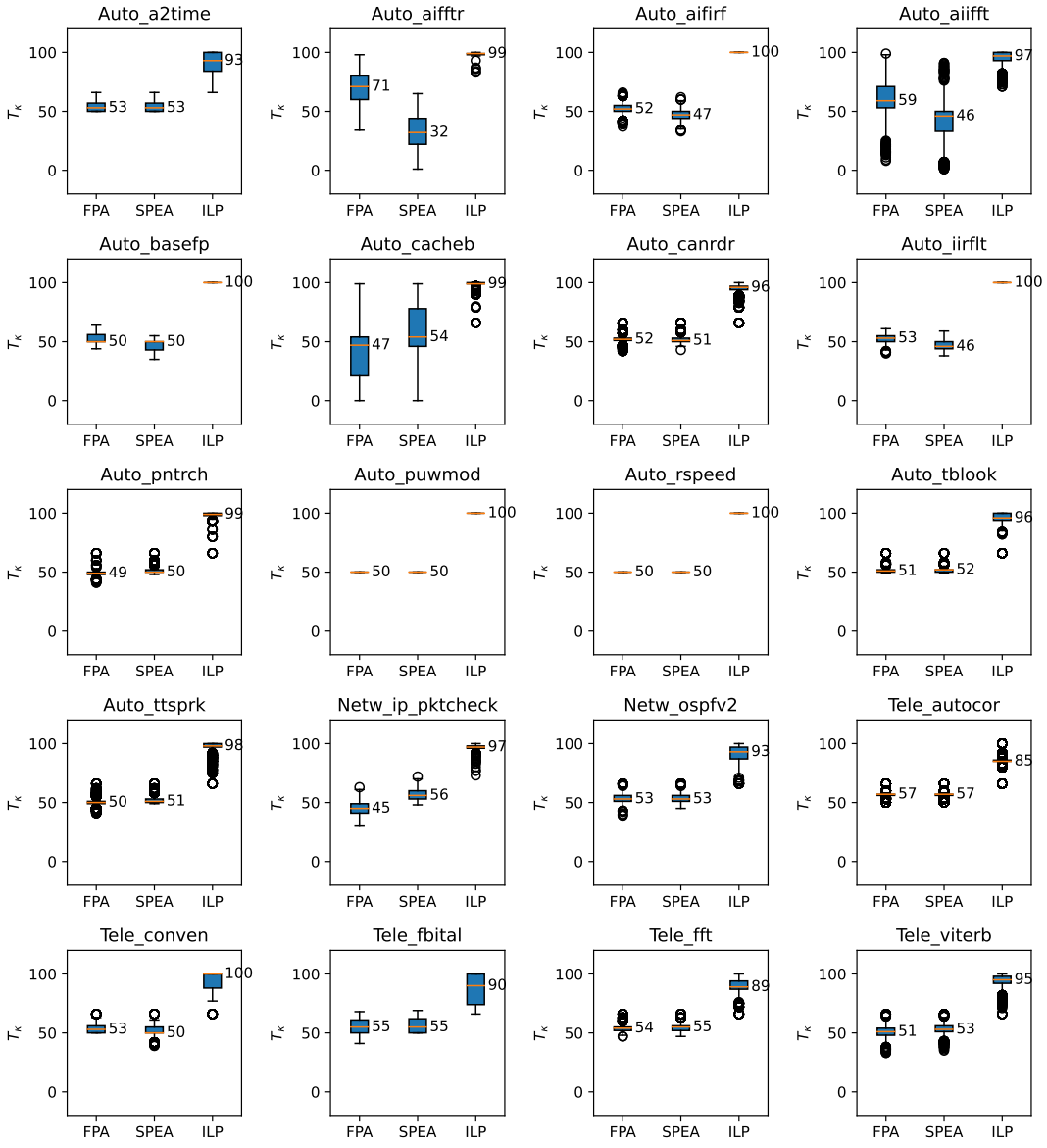
$$T_\kappa = \left(1 - \frac{\mathcal{T}'_\kappa}{\sum_\kappa \mathcal{T}'_\kappa}\right) * 100 \tag{9.19}$$

where κ is the respective method and $T_\kappa \in [0, 100]$. This is a standard way to normalize the value of a parameter (in our case, $\mathcal{T}'_\kappa$) between the range $[0, 100]$. The T_κ metric quantifies whether method $\kappa \in \{F, S, I\}$—FPA, SPEA, or ILP—is better relative to the other considered methods. The higher the value of T_κ, the better the solution quality.

Let T_F, T_S, and T_I be the relative execution metrics for FPA, SPEA, and ILP, respectively. We calculate the total DMA execution times $\mathcal{T}'_F$, $\mathcal{T}'_S$, and $\mathcal{T}'_I$ for each individual during the MO_{DMA} run, and using these values, we calculate the relative execution metrics T_F, T_S, and T_I for each individual. After the MO_{DMA} optimization is finished, we have the set of relative execution metrics from every individual generated during the optimization for each method. To decide the best method to perform DCPO, we compare the median values of the sets of T_F, T_S, and T_I metrics. The higher the median of the relative execution metric, the better the respective optimization method for DCPO.

Figure 9.12 shows the box plots for all benchmarks, where the relative execution metrics T_κ for FPA, SPEA, and ILP for all individuals from MO_{DMA} are plotted. The x-axis of every sub-figure represents the three methods—FPA, SPEA, and ILP—used to solve the DCPO optimization, the y-axis of every sub-figure represents the relative execution metric, and the median value of each method is indicated within every box plot. The boxes within the figure represent the interquartile range (IQR), which contains the central 50% of the

Figure 9.12.: Comparison between the relative execution metrics (T_K) for FPA-, SPEA-, and ILP-based DCPO



data points. The line inside the box indicates the median, showing the central tendency of the data. The average T_K value can skew due to outliers, so we use the median values. The whiskers extend to the minimum and the maximum values that fall within 1.5 times the IQR from the lower and upper quartiles. If there are any T_K values outside the whiskers, they are considered outliers in the box plot, representing unusually high or low values relative to the rest of the data.

From overall evaluations, we can see that DCPO–ILP performed the best for all benchmarks compared to the FPA- and SPEA-based approaches. For 7 benchmarks, the SPEA-based approach performed better than the FPA-based

approach, and for 6 benchmarks, FPA performed better than SPEA. For the remaining 7 benchmarks, the FPA- and SPEA-based approaches provided the same median value for the T_K metric. Therefore, the SPEA algorithm performed slightly better than FPA during the evaluation run.

Furthermore, we compared the compilation time taken by FPA, SPEA, and ILP methods to solve the DCPO problem. On average, FPA and SPEA took 9.64 s and 12.58 s to solve the DCPO problem, whereas ILP-based DCPO took 11.19 min. The average compilation time for ILP-based DCPO is higher than FPA and SPEA due to `Auto_aifftr`, `Auto_aifirf`, and `Auto_iirflt`, where they took, on average, 73.71 min. For the remaining 17 benchmarks, the ILP-based approach took 9.73 s, on average, to solve the DCPO problem. In the case of FPA and SPEA, `Auto_aifftr` took the most compilation time to solve the DCPO problem—1.95 min and 2.4 min, respectively.

We also compared the solutions obtained from FPA-, SPEA-, and ILP-based DCPO with the solution that performs all DMA transfers in the burst mode for the respective decision vector x. On average, FPA and SPEA methods showed a 19.03% and 20.21% decrease in the total execution time ($\mathcal{T}'(y)$) compared to the burst mode ($\mathcal{T}(x)$). Consequently, we can say that SPEA performed slightly better than FPA in terms of achieved reduction in $\mathcal{T}'(y)$. Moreover, ILP-based DCPO provided a 28.15% decrease in the total execution time, on average, using arbitration mode compared to the burst mode solutions. Therefore, we can conclude that using the arbitration mode and DCPO is more advantageous than performing DMA transfer in the burst mode. From an overall comparison, we can say that the deterministic ILP-based approach is the best for solving the single-objective DCPO optimization.

9.5.3 Optimizations Comparison with 60% Relative SPM Size and DCPO–FPA

In this subsection, we consider 60% ISPM and DSPM size relative to the benchmark size, perform compiler-level MO_{DMA}, MO_S, MO_D, SO_D, and SO_{DMA} optimizations, and compare all the optimization runs in terms of the quality of the obtained solutions. Moreover, we compare the MO_{DMA} solutions with the baseline evaluation BL, where the whole code is executed from the Flash. MO_{DMA}, MO_S, and MO_D simultaneously minimize WCET and energy consumption and provide final Pareto optimal solution sets. SO_D and SO_{DMA} are single-objective optimizations that minimize WCET, so we perform energy analysis for their final solutions to obtain respective energy values. Therefore, in the case of SO_D and SO_{DMA}, we will always have one solution on their respective Pareto fronts.

As DCPO–FPA, on average, provided the worst solution quality in Subsection 9.5.2 compared to the SPEA and ILP methods, we use it to evaluate the performance of MO_{DMA} in the worst case against SO_D, SO_{DMA}, MO_S, and MO_D in this subsection. The worst-performing DCPO–FPA will provide

us with the worst possible DMA call placements in one-to-one individual comparisons to the ILP and SPEA methods. Therefore, if the combination of DCPO–FPA and MO_{DMA} provides better solutions than the previously proposed static and dynamic allocation approaches, we can, by extension, say that any combination of the DCPO method and MO_{DMA} will also provide better solutions.

9.5.3.1 Pareto Fronts

Obtaining the true Pareto front for such a multi-objective optimization problem is ambitious. So, realistically assuming that the true Pareto front is unknown, we construct the final Pareto front (PF_T) for each benchmark from the union of solutions obtained from all the optimization runs. During this evaluation, PF_F is the final Pareto front obtained at the end of each optimization run. PF_T is the best available approximation of the true Pareto front obtained from the union of all PF_Fs from MO_{DMA}–FPA, MO_{DMA}–SPEA, MO_S–FPA, MO_S–SPEA, MO_D–FPA, MO_D–SPEA, SO_D–ILP, SO_{DMA}–ILP, and BL, against which we evaluate the quality of solutions for each optimization.

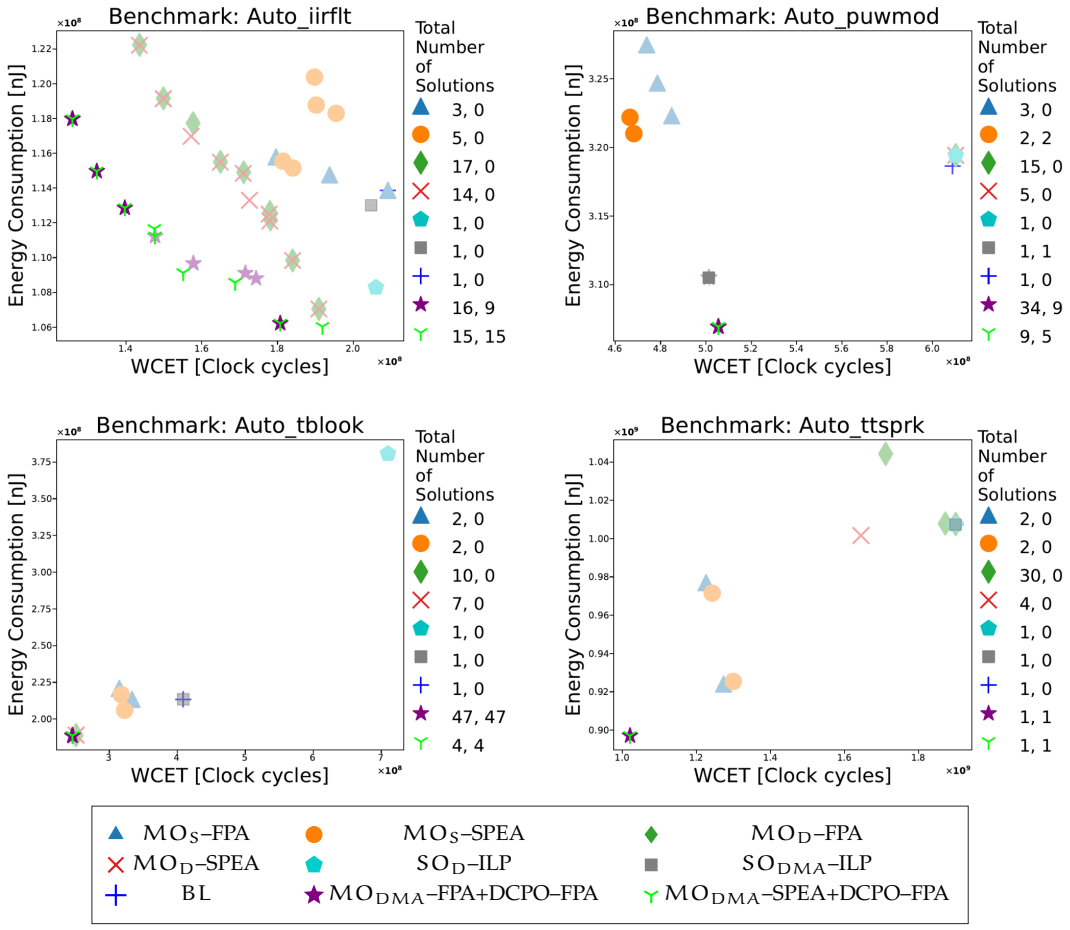
Figure 9.13 shows the solutions found by MO_{DMA}+DCPO–FPA, MO_S, and MO_D using FPA and SPEA, SO_D and SO_{DMA} using ILPs, and the BL evaluation. For brevity's sake, we present the solutions for only four benchmarks. Figures E.1 and E.2 in Appendix E present the Pareto optimal fronts for all remaining benchmarks from these evaluations, using the 60% SPM size and DCPO–FPA configuration.

In Figure 9.13, we can see 2D scatter plots for Pareto fronts obtained for four benchmarks—`Auto_iirflt`, `Auto_puwmod`, `Auto_tblook`, and `Auto_ttsprk`. The title of each subplot is the name of the respective benchmark. The legend at the bottom of the figure presents the optimization run to which a particular solution belongs, and the legend at the right side of each subplot shows the total number of solutions on PF_F and PF_T for each optimization run. The PF_T for each benchmark is represented with the markers and their respective colors from the legend, and the fainter version of those colors represents the respective Pareto fronts PF_F returned by each approach.

For the `Auto_iirflt` benchmark, all solutions on PF_T are from MO_{DMA}–FPA and MO_{DMA}–SPEA. However, the PF_F of MO_{DMA}–FPA has some solutions that do not lie on the final PF_T. In the case of the `Auto_puwmod` benchmark, MO_{DMA}–FPA, MO_{DMA}–SPEA, and SO_{DMA} optimization runs contributed solutions to the final PF_T. MO_{DMA}–FPA, in this case, contributed the largest number of solutions to the final PF_T.

For the `Auto_tblook` benchmark, MO_{DMA}–FPA and MO_{DMA}–SPEA have 47 and 4 solutions on their respective PF_Fs, and all of them lie on the final PF_T. None of the solutions provided by other optimizations lies on the final PF_T. In the case of the `Auto_ttsprk` benchmark, MO_{DMA}–FPA and MO_{DMA}–SPEA found one solution that lies on the final PF_T.

Figure 9.13.: Pareto front for four benchmarks with 60% relative SPM size and MO_{DMA}+DCPO–FPA configuration



In some cases, multiple solutions might be mapped to the same point in objective space. Similar to the case discussed in Subsection 8.5.2, all MO_{DMA}–FPA and MO_{DMA}–SPEA solutions for `Auto_tblook` are different in the solution space due to their placement of different global data objects in DSPM. However, none of the global data objects impacted the objective values. Consequently, all the MO_{DMA} solutions converged to a single point in the objective space. Moreover, we already discussed the `Auto_tblook`'s solution obtained using SO_D–ILP in Subsection 8.5.2. Therefore, we skip the discussion about SO_D–ILP solutions for brevity's sake.

Furthermore, in the case of `Auto_puwmod`, we found 34 and 9 solutions using MO_{DMA}–FPA and MO_{DMA}–SPEA, respectively, mapped to two points in the objective space. In the case of `Auto_iirflt`, MO_{DMA}–FPA found 16 solutions that are mapped to 8 points in the figure as its PF_F, and MO_{DMA}–SPEA found 15 solutions that are mapped to 9 points in the figure as its PF_F.

From overall evaluations, we calculated the total number of solutions provided by each optimization run on the final PF_T and averaged it over all the

Table 9.1.: Comparison between the average number of solutions (in %) on the final PF_T from respective PF_F of each optimization approach for 60% relative SPM size and MO_{DMA}+DCPO–FPA configuration

MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
8.2	42.62	0	0	5	10	73.8	71.59	0

benchmarks. Table 9.1 provides the percentage of the number of solutions on PF_T on average out of the solutions on PF_F for each optimization run. None of the solutions found by MO_D and BL lie on the final PF_T. The MO_S optimization run, on average, provided some percentage of solutions on the final PF_T, and the SPEA algorithm provided more solutions on PF_T compared to FPA in this case. Only for the `Auto_basefp` benchmark, SO_D–ILP found the solution of its final PF_T. For `Auto_rspeed`, the solution obtained by SO_{DMA} lies on the final PF_T along with the solutions obtained by MO_S–SPEA, MO_{DMA}–FPA, and MO_{DMA}–SPEA. SO_{DMA} was also able to find a solution for `Auto_puwmod` on the final PF_T (cf. Figure 9.13).

SO_{DMA} performs function-level dynamic allocation and uses the DMA controller in burst mode to perform DMA transfer. As mentioned in Subsection 8.5.2, all the `EEMBC` benchmarks considered during evaluations have a large function, "`t_run_test()`", which does not fit completely in the ISPM. Consequently, SO_{DMA} could not place this function within ISPM. Other than the `t_run_test()` function, the considered benchmarks have smaller functions that can be placed within ISPM by SO_{DMA}. However, the extra WCET and energy costs needed for performing dynamic allocations might be more than the benefits of allocating these functions within the ISPM. Therefore, although SO_{DMA} aimed to minimize WCET, it did not provide great results for most `EEMBC` benchmarks. On the other hand, MO_{DMA} considers both functions and loops, leading to much better results than SO_{DMA}.

As mentioned in Sections 5.4.2, 6.5.3, and 8.5.2, Table 9.1 can skew numbers towards single-solution-producing approaches. Therefore, we can see a 5% contribution from SO_D–ILP, even though it provided one solution to PF_T for one benchmark, and 10% for SO_{DMA}, even though it provided two solutions on PF_T only for two benchmarks. On average, MO_{DMA} clearly provided the most solutions on the final PF_T, and the FPA algorithm contributed slightly more solutions on PF_T than SPEA for MO_{DMA}.

As mentioned before, we need to perform various code transformations at the assembly level to facilitate the dynamic allocation of memory objects during runtime. We also need to add DMA-related code and DMA transfer calls to generate the DMA-aware dynamically allocatable code. We averaged the code size of all the MO_{DMA} solutions for all the benchmarks and compared it with the BL solution's code size. MO_{DMA}–FPA and MO_{DMA}–SPEA showed an average increase in code size by 46.89% and 44.79%, respectively, compared

Table 9.2.: Average reduction in WCET and energy consumption objectives (in %) obtained using MO_S, MO_D, SO_D, SO_{DMA}, and MO_{DMA}+DCPO–FPA over the BL evaluation run

Objectives	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}	
	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA
WCET	38.36	38.58	36.7	36.74	6.37	1.42	37.41	37.46
Energy consumption	14	14.34	11.26	11.49	-0.65	0.42	9.83	9.8

to the BL's average code size, i.e., MO_{DMA} caused an average 45.92% increase in code size compared to the original benchmark size.

We also compared the reduction in WCET and energy consumption objectives achieved using MO_{DMA}, MO_D, MO_S, SO_D, and SO_{DMA} relative to the BL evaluation. Table 9.2 shows the percentage average reduction in objective values obtained using the optimization approaches over BL. In Table 9.2, percentage reductions are computed as the reduction of average objective values across all benchmarks, relative to the BL evaluations. From Table 9.2, we observe that the reduction in the WCET objective is, on average, higher than that in energy consumption. MO_S achieved a better average reduction in WCET and energy consumption than MO_{DMA}. However, MO_S provided fewer solutions on the final PF_T than MO_{DMA}. Nevertheless, the reduction in objective value for MO_S averaged it to a slightly higher value than MO_{DMA}. Furthermore, as mentioned before, SO_D generated solutions have worse WCET and energy consumption than BL for some benchmarks. Consequently, Table 9.2 shows a slight average increase in energy consumption for SO_D compared to the BL evaluation.

Lastly, we compared MO_{DMA} with MO_D to assess the performance of our proposed approach against the previously proposed multi-objective dynamic allocation. Unlike the aggregated reductions reported in Table 9.2, we perform a per-benchmark comparison between MO_{DMA} and MO_D. For each benchmark, we first computed the percentage reduction in WCET and energy consumption, and then averaged these reductions across all benchmarks. Using this approach, we obtained an average reduction of 20.34% in WCET and 7.8% in energy consumption with MO_{DMA} compared to MO_D.

We can conclude from Table 9.1, Table 9.2, Figure 9.13, Figure E.1, and Figure E.2 the following:

- The single-objective optimization approaches are, in general, unable to provide good-quality solutions on the final Pareto front.
- MO_{DMA} always provided better quality solutions than the BL evaluation.
- MO_D from Chapter 8 is outperformed by the MO_{DMA} approach proposed in this chapter due to the large overheads of CPU-based dynamic copying of memory objects.

Table 9.3.: Total number of benchmarks with best (or nondominated) quality indicators for each optimization approach for 60% relative SPM size and MO_{DMA}+DCPO–FPA configuration

Quality	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
Indicators	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
$\mathcal{C}$	0 (2)	3 (7)	0 (0)	0 (0)	0 (1)	0 (2)	1 (11)	2 (9)	0 (0)
NR	0 (1)	5 (2)	0 (0)	0 (0)	0 (0)	0 (0)	6 (3)	4 (4)	0 (0)
RNS	0 (2)	3 (7)	0 (0)	0 (0)	0 (1)	0 (2)	1 (11)	2 (9)	0 (0)
HV	1 (1)	6 (1)	0 (0)	0 (0)	1 (0)	0 (0)	0 (8)	3 (8)	0 (0)

- The static allocation-based multi-objective optimization (MO_S) occasionally produces solutions of good quality on the Pareto front. However, the proposed MO_{DMA}, in general, generates a much larger variation of final Pareto optimal solutions.

9.5.3.2 Quality Indicators

To better analyze these trends, we consider four quality indicators—*Coverage* ($\mathcal{C}$) (cf. Definition 4.6), *Non-dominance Ratio* (NR) (cf. Definition 4.7), *Ratio of Non-dominated Solutions* (RNS) (cf. Definition 4.8), and *HyperVolume* (HV) (cf. Definition 4.9)—and compare all the optimizations using them. For reference sake, Table E.1 and Table E.2 in Appendix E present the quality indicator $\mathcal{C}$, NR, RNS, and HV values, respectively, for all the benchmarks for these evaluations with 60% SPM size and DCPO–FPA configuration.

Table 9.3 summarizes the total number of benchmarks with best or nondominated $\mathcal{C}$, NR, RNS, and HV for each approach. In this table, the numbers shown outside the bracket represent the total number of benchmarks with best quality indicators, and the numbers inside the bracket represent the number of benchmarks with nondominated Pareto optimal solutions compared to other approaches. For example, the final PF_Ts of `Auto_iirflt`, `Auto_tblook`, and `Auto_ttsprk` benchmarks have nondominated Pareto optimal solutions from MO_{DMA}–FPA and MO_{DMA}–SPEA, and `Auto_puwmod` has nondominated solutions from MO_{DMA}–FPA, MO_{DMA}–SPEA, MO_S–SPEA, and SO_{DMA} (cf. Figure 9.13). In these cases, we cannot conclude the best approach as multiple approaches provided solutions on the final PF_T. For the `Auto_cacheb` benchmark, MO_{DMA}–SPEA found six solutions on its PF_F, out of which five solutions were on the final PF_T (cf. Figure E.2). Therefore, in this case, we can say that MO_{DMA}–SPEA performed best compared to other approaches.

Table 9.3 shows that, on average, MO_{DMA} provided better quality solutions compared to previous approaches. MO_D and BL provided no solutions on the final PF_T for any benchmark. SO_D provided the best solution in terms of HV and nondominated solution in terms of $\mathcal{C}$ and RNS for one benchmark. SO_{DMA} provided a nondominated solution in terms of $\mathcal{C}$ and RNS for two benchmarks.

For some benchmarks, MO_S provided the best or nondominated solutions on PF_T. For `Tele_conven`, only MO_S–FPA and MO_S–SPEA provided a single solution each on the final PF_T—for both MO_S–FPA and MO_S–SPEA, all quality metrics are indifferent. For `Tele_autocor`, only MO_S–FPA and MO_S–SPEA contributed all the solutions to the final PF_T. However, MO_S–SPEA contributed more solutions to the final PF_T than MO_S–FPA in the case of `Tele_autocor`, indicated by the superior `NR` metric performance.

For `Auto_aifftr`, both MO_S–FPA and MO_S–SPEA contributed to the final PF_T. However, MO_S–SPEA outperformed MO_S–FPA by contributing more solutions from its PF_F to the final PF_T, so MO_S–SPEA performed better in terms of all four quality indicators than MO_S–FPA.

As mentioned in Subsection 8.5.2, some benchmarks in `EEMBC` have functions and loops bigger than ISPM size, making basic block-level MO_S more effective. Therefore, a system designer can choose to use MO_{DMA} if the constraint condition in Equation 8.10 is satisfied. Equation 8.10 implies that the sum of the sizes of the memory objects smaller than ISPM size must be greater than or equal to ISPM size. If this condition is not satisfied, MO_S might provide better-quality solutions.

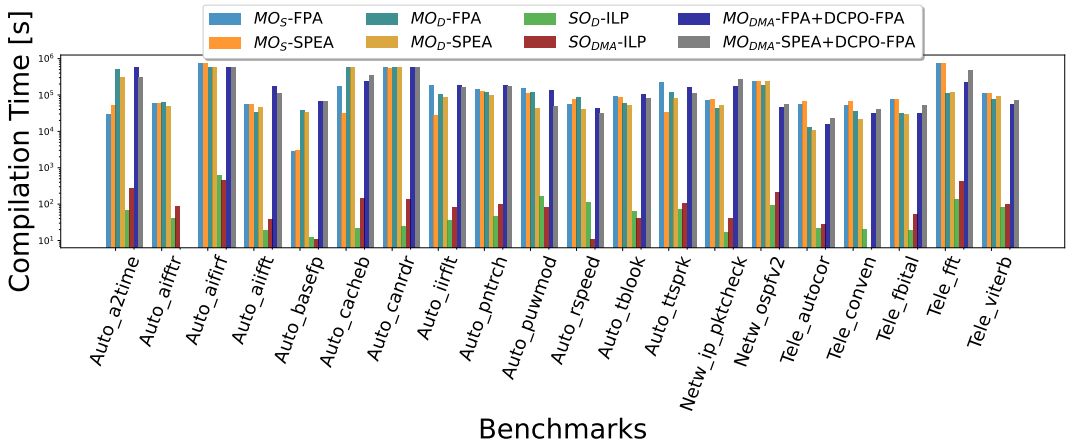
For two benchmarks—`Auto_a2time` and `Auto_puwmod`—, MO_{DMA}–FPA, MO_{DMA}–SPEA, and MO_S–SPEA provided nondominated solutions on the final PF_T concerning $\mathcal{C}$ and `RNS`. However, MO_{DMA}–FPA outperformed the other two approaches in terms of the `NR` metric by providing the highest number of solutions to the final PF_T. Moreover, MO_{DMA}–FPA and MO_{DMA}–SPEA both provided nondominated solutions in terms of the `HV` metric.

For `Auto_rspeed`, MO_{DMA}–FPA, MO_{DMA}–SPEA, SO_{DMA}, and MO_S–SPEA provided nondominated solutions in terms of $\mathcal{C}$ and `RNS`. MO_S–SPEA had the best `HV` metric value, and MO_{DMA}–FPA provided the largest number of solutions with the best `NR` value. For brevity's sake, we skip the explanation for the remaining benchmarks.

From overall evaluations and Table 9.3, we can see that MO_{DMA} clearly outperformed all other approaches. For the 11 and 9 benchmarks, MO_{DMA}–FPA and MO_{DMA}–SPEA have nondominated performance in terms of $\mathcal{C}$ and `RNS`, with 3 and 4 benchmarks having nondominated `NR` metrics, respectively. Moreover, for 8 benchmarks, MO_{DMA}–FPA and MO_{DMA}–SPEA have nondominated `HV` metrics. Finally, for 3 benchmarks, MO_{DMA}–SPEA provided the best solutions in terms of the `HV` metric.

MO_{DMA}–FPA and MO_{DMA}–SPEA each provided the best solutions in terms of $\mathcal{C}$ and `RNS` for 1 and 2 benchmarks, respectively. Moreover, MO_{DMA}–FPA and MO_{DMA}–SPEA have the best `NR` metrics for 6 and 4 benchmarks, respectively. When compared to the rest of the approaches, the proposed MO_{DMA} approach is clearly better when Equation 8.10 is satisfied.

Figure 9.14.: Compilation times for MO_S, MO_D, SO_D, SO_{DMA}, and MO_{DMA}+DCPO–FPA runs with 60% SPM size configuration



9.5.3.3 Compilation Times

In this subsection, we compare the compilation time required to solve the MO_{DMA}+DCPO–FPA, MO_D, MO_S, SO_{DMA}, and SO_D optimizations with each other. Figure 9.14 presents the compilation times required by these five optimization approaches in the form of a bar plot for the 60% relative SPM size configuration. The x-axis represents all the evaluated benchmarks from `EEMBC`, and the y-axis shows the compilation time required to finish the optimizations. From the figure, we can see that the multi-objective MO_{DMA}, MO_D, and MO_S optimizations are more time-consuming than the single-objective SO_D and SO_{DMA} optimizations during this evaluation.

SO_{DMA} and SO_D, on average, took 2.12 min and 1.4 min to complete the optimization runs, respectively. MO_S–FPA and MO_S–SPEA, on average, took 52.68 h and 45.48 h to complete the optimization runs, respectively. MO_D–FPA and MO_D–SPEA took, on average, 48.89 h and 44.07 h to complete the optimization runs, respectively. MO_{DMA}–FPA and MO_{DMA}–SPEA took, on average, 69.72 h and 69.11 h to complete the optimization runs, respectively.

In Subsection 8.5.1, we calculated the maximum number of WCET and energy evaluations to complete the multi-objective optimization using metaheuristic algorithms. Therefore, as MO_{DMA}, MO_D, and MO_S optimization runs require a maximum of 10000 evaluations, they are more time-consuming than the single-objective optimizations during these evaluations. As SO_D and SO_{DMA} need to perform WCET and energy analyses twice to build their ILP models and solve their respective ILPs once, they were faster than MO_{DMA}, MO_D, and MO_S at the detriment of SO_D and SO_{DMA}'s solution quality. Moreover, MO_{DMA} needs to perform not only WCET and energy analyses but also solve the address assignment problems and perform DCPO multiple times.

Table 9.4.: Comparison between the average number of solutions (in %) on the final PF_T from respective PF_F of each optimization approach for 60% relative SPM size and MO_{DMA}+DCPO–ILP configuration

MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
8.2	42.62	0	0	5	5.26	86.75	78.57	0

Therefore, MO_{DMA} took more time than all other considered optimizations. Consequently, MO_{DMA}+DCPO–FPA, on average, took 33.04% and 29.29% more compilation time compared to MO_D and MO_S, respectively.

9.5.4 Optimizations Comparison with 60% Relative SPM Size and DCPO–ILP

In this subsection, we consider 60% ISPM and DSPM size relative to the benchmark size and solve the MO_{DMA} problem using DCPO–ILP instead of DCPO–FPA. We compare the solutions obtained from the MO_{DMA}+DCPO–ILP optimization run with solutions obtained from all other approaches.

9.5.4.1 Pareto Fronts

Figure 9.15 shows the solutions found by MO_{DMA}+DCPO–ILP for four benchmarks—`Auto_iirflt`, `Auto_puwmod`, `Auto_tblook`, and `Auto_ttsprk`—and compares them with the solutions from all other approaches. Figures E.3 and E.4 in Appendix E present the Pareto fronts for all remaining benchmarks from these evaluations, using the 60% SPM size and DCPO–ILP configuration.

Comparing Figures 9.15 and 9.13 for the `Auto_iirflt` benchmark, we can see that MO_{DMA}–FPA+DCPO–FPA found more solutions on both PF_F and PF_T. For `Auto_puwmod` and `Auto_tblook`, MO_{DMA}+DCPO–ILP found the same or more solutions on both PF_F and the final Pareto front PF_T. For `Auto_ttsprk`, MO_{DMA}–FPA+DCPO–ILP was the only optimization that found the single solution on the final PF_T. Due to the random nature of metaheuristic algorithms, it is impossible to predict the shape and composition of the final Pareto front. It is evident from Figures 9.15 and 9.13 that, for the same benchmarks, we got different-looking Pareto fronts. However, the trend that MO_{DMA} performed better than other approaches for most benchmarks remained the same.

Table 9.4 provides a comparison between the average number of solutions obtained by each approach on the final PF_T from the respective PF_F. MO_{DMA}+DCPO–ILP has more solutions on the final PF_T than in Subsection 9.5.3 (cf. Table 9.1). Moreover, MO_D and BL were not able to contribute any solutions to the final Pareto front PF_T.

Table 9.5 provides the average percentage reduction in objective values obtained using the optimization approaches over the BL evaluation. As

Figure 9.15.: Pareto front for four benchmarks with 60% relative SPM size and MO_{DMA}+DCPO–ILP configuration

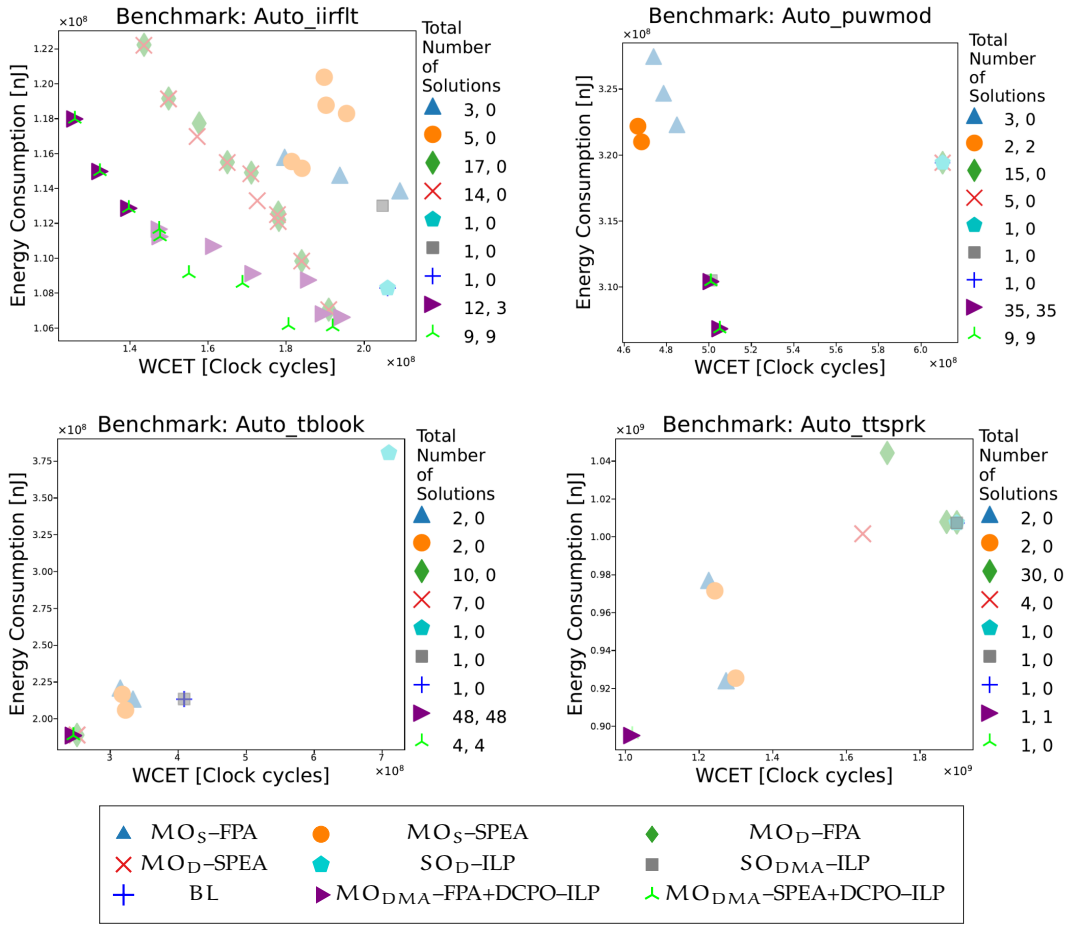


Table 9.5.: Average reduction in WCET and energy consumption objectives (in %) obtained using MO_S, MO_D, SO_D, SO_{DMA}, and MO_{DMA}+DCPO–ILP over the BL evaluation run

Objectives	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}	
	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA
WCET	38.36	38.58	36.7	36.74	6.37	1.42	37.81	37.71
Energy consumption	14	14.34	11.26	11.49	-0.65	0.42	11.82	11.97

in Subsection 9.5.3.1, the reductions reported in Table 9.5 are computed as reductions of average objective values across all benchmarks, relative to BL. Similar to Subsection 9.5.3, MO_S achieved a better average reduction in WCET and energy consumption than MO_{DMA}. However, MO_S provided fewer solutions on PF_T than MO_{DMA}. Moreover, from Tables 9.5 and 9.2, we can see that MO_{DMA}+DCPO–ILP achieved a slightly higher average reduction in WCET and energy consumption objectives compared to MO_{DMA}+DCPO–FPA.

Table 9.6.: Total number of benchmarks with best (or nondominated) quality indicators for each optimization approach for 60% relative SPM size and MO_{DMA}+DCPO–ILP configuration

Quality	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
Indicators	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
$\mathcal{C}$	0 (2)	4 (5)	0 (0)	0 (0)	0 (1)	0 (1)	2 (10)	3 (10)	0 (0)
NR	0 (1)	5 (2)	0 (0)	0 (0)	0 (0)	0 (0)	8 (2)	3 (3)	0 (0)
RNS	0 (2)	4 (5)	0 (0)	0 (0)	0 (1)	0 (1)	2 (10)	3 (10)	0 (0)
HV	1 (1)	6 (1)	0 (0)	0 (0)	1 (0)	0 (0)	1 (8)	2 (8)	0 (0)

Furthermore, we compared MO_{DMA}+DCPO–ILP with MO_D to determine the performance of our proposed approach against the previously proposed MO_D. Following the per-benchmark comparison methodology mentioned in Subsection 9.5.3.1, we evaluate the performance of MO_{DMA}+DCPO–ILP relative to MO_D. We achieved, on average, 19.17% and 7.64% reductions in the WCET and energy consumption objectives, respectively, compared to MO_D. Overall, MO_{DMA}+DCPO–ILP provided slightly better solutions in this subsection compared to the MO_{DMA}+DCPO–FPA optimization run from Subsection 9.5.3.

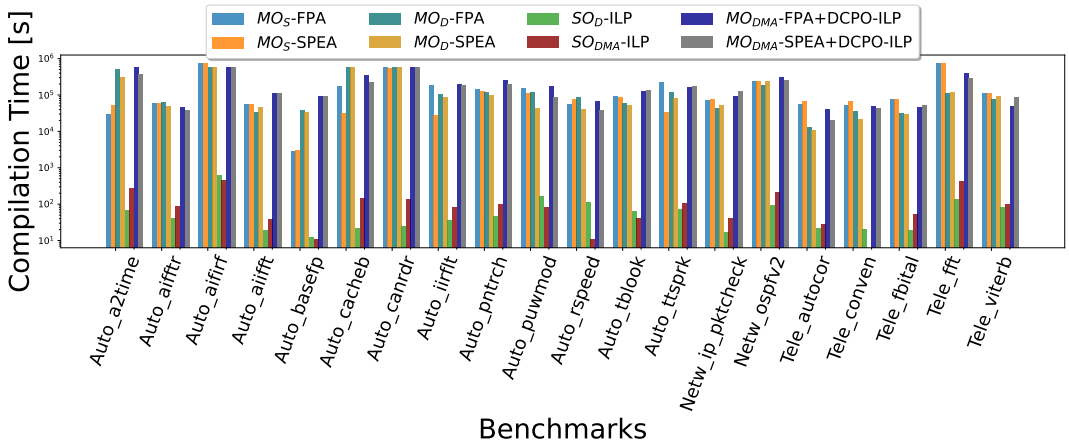
9.5.4.2 Quality Indicators

Table 9.6 provides the total number of benchmarks with best or nondominated Coverage ($\mathcal{C}$), Non-dominance Ratio (NR), Ratio of Non-dominated Solutions (RNS), and HyperVolume (HV) metrics for each approach. For reference sake, Tables E.3 and E.4 in Appendix E present the quality indicator $\mathcal{C}$, NR, RNS, and HV values, respectively, for all benchmarks from these evaluations, using the 60% SPM size and DCPO–ILP configuration.

Comparing Tables 9.6 and 9.3, we see variations in the number of best and nondominated quality indicators. Overall, MO_{DMA}+DCPO–ILP provided slightly better solutions compared to the MO_{DMA}+DCPO–FPA run.

MO_{DMA}+DCPO–ILP performed better than other approaches with the best or nondominated quality indicators for most benchmarks. MO_{DMA}–SPEA performed best for 3 benchmarks in terms of $\mathcal{C}$, NR, and RNS metrics and for 2 benchmarks in terms of the HV metric. MO_{DMA}–FPA performed best for 8 benchmarks in terms of the NR quality indicator. MO_S–SPEA provided 6 benchmarks with a better HV metric. However, overall MO_{DMA} had better or nondominated HV values for more benchmarks. MO_{DMA}–SPEA performed slightly well compared to MO_{DMA}–FPA in terms of the HV metric. Overall, MO_{DMA}–SPEA performed better in terms of $\mathcal{C}$ and RNS, and MO_{DMA}–FPA performed better in terms of NR, i.e., MO_{DMA}–SPEA had a better ratio of solutions from individual Pareto fronts PF_Fs that ended up on the final PF_T, and MO_{DMA}–FPA had most (86.75%) solutions on the final PF_T.

Figure 9.16.: Compilation times for MO_S, MO_D, SO_D, SO_{DMA}, and MO_{DMA}+DCPO–ILP runs with 60% SPM size configuration



9.5.4.3 Compilation Times

We also compare the compilation time required by these approaches. Figure 9.16 provides the compilation times required by the MO_{DMA}+DCPO–ILP, MO_D, MO_S, SO_{DMA}, and SO_D optimization approaches in the form of a bar plot for the 60% relative SPM size configuration.

Similar to Subsection 9.5.3, in this subsection, SO_D and SO_{DMA} provided poor solution quality but were extremely fast compared to MO_{DMA}. Moreover, MO_{DMA}+DCPO–ILP, on average, needed 16.07% and 11.37% more compilation time than MO_D and MO_S optimization runs, respectively. Furthermore, MO_{DMA}+DCPO–ILP, on average, required 20.22% less compilation time than MO_{DMA}+DCPO–FPA.

Solving the DCPO problem contributes to the total compilation time for MO_{DMA}. Moreover, from Subsection 9.5.2, we can see that DCPO–ILP required more time compared to DCPO–FPA. However, from Subsections 9.5.3 and 9.5.4, we can see that the MO_{DMA}+DCPO–ILP evaluation run was faster than the MO_{DMA}+DCPO–FPA run. Therefore, we could deduce that the time taken to solve the DCPO problem is not the major contributing factor for MO_{DMA}'s compilation time.

Generally, the compilation time required by MO_S, MO_D, and MO_{DMA} largely depends on the time taken by `aiT` and `EnergyAnalyzer` to perform respective analyses, which can vary depending on the individuals considered for analyses. As metaheuristic algorithms are non-deterministic, it becomes difficult to derive and conclude trends for compilation times for these multi-objective optimizations.

From the evaluations conducted in Subsections 9.5.2, 9.5.3, and 9.5.4, we can conclude the following:

- If DCPO solutions are compared one-to-one, ILPs will provide optimal solutions compared to FPA and SPEA, on average.
- Due to the random nature of the metaheuristic algorithms, it is difficult to get the same solutions consistently on the final PF_T. However, MO_{DMA} with both the worst-performing DCPO–FPA and the best-performing DCPO–ILP provides better solutions compared to all other approaches, except when Equation 8.10 is not satisfied, MO_S could give better solutions.
- MO_{DMA} provided a significant reduction in WCET and energy consumption, especially compared to the BL evaluation.
- MO_{DMA} outperformed all previous dynamic allocation-based optimization approaches.
- Using the DMA controller instead of the CPU to perform dynamic allocation during runtime clearly reduces the WCET and energy consumption of the program under consideration.

9.5.5 Optimizations Comparison with 20%, 40%, and 80% Relative SPM Size

In Subsections 9.5.3 and 9.5.4, we considered 60% ISPM and DSPM sizes relative to the benchmark size, and we evaluated and compared all previous optimizations with the proposed MO_{DMA}. There, we used DCPO–FPA and DCPO–ILP to find optimal places within the code to place DMA calls, respectively. In this subsection, we evaluate the effect of different SPM sizes on the proposed optimization. Consequently, we consider three different configurations of SPM sizes—20%, 40%, and 80%—relative to the benchmark size. Furthermore, for the sake of brevity, we only compare all previous optimizations with the MO_{DMA} optimization in conjunction with DCPO–FPA.

For reference sake, Tables E.5 and E.6 in Appendix E present the quality indicator $\mathcal{C}$, NR, RNS, and HV values, respectively, for all benchmarks from these evaluations, using the 20% SPM size and DCPO–FPA configuration. Similarly, Tables E.7 and E.8 present quality indicator values for the 40% SPM size configuration, and Tables E.9 and E.10 present quality indicator values for the 80% SPM size configuration. Moreover, Figures E.5, E.6, and E.7 in Appendix E show the Pareto fronts for all benchmarks for the 20% SPM size configuration; Figures E.8, E.9, and E.10 show the Pareto fronts for the 40% SPM size configuration; Figures E.11, E.12, and E.13 show the Pareto fronts for the 80% SPM size configuration.

Table 9.7 provides the percentage average reduction in the objective values obtained using the optimization approaches over the BL run for the 20%, 40%,

Table 9.7.: Reduction of average WCET and energy consumption (in %) obtained using MO_S, MO_D, SO_D, SO_{DMA}, and MO_{DMA}+DCPO–FPA relative to BL, for 20%, 40%, and 80% relative SPM size configuration

Relative SPM Size(%)	Objectives	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}	
		FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA
20%	WCET	22.34	23.43	32.89	32.84	1.01	-4.53	33.38	33.44
	Energy consumption	8.36	9.1	10.23	10.25	-2.77	-4.69	10.46	10.77
40%	WCET	36.82	37.07	36.25	37.6	6.36	-3.19	37.8	37.69
	Energy consumption	11.6	11.9	13.49	13.7	-0.65	-3.27	10.07	10.23
80%	WCET	20.95	21.01	36.91	37.01	9.63	-3.16	37.99	37.86
	Energy consumption	7.75	7.82	11.61	12.01	0.85	-3.64	10.15	10.31

Table 9.8.: Average per-benchmark reduction in WCET and energy consumption (in %) obtained using MO_{DMA} over MO_D, for 20%, 40%, and 80% relative SPM size configuration

Objectives	Relative SPM sizes (in %)		
	20%	40%	80%
WCET	15.92	18.39	21.52
Energy consumption	4.44	7.5	6.73

and 80% relative SPM size cases. The reductions reported in Table 9.7 are computed as reductions of average objective values across all benchmarks, relative to BL. From this table, we can see that we achieved a slightly higher reduction in WCET and energy consumption using MO_{DMA} compared to the other optimization approaches for all the SPM size configurations. Furthermore, MO_{DMA} still achieves substantial reductions in objective values even for a tiny 20% relative SPM size.

Moreover, Table 9.8 shows the percentage reduction in WCET and energy consumption obtained, on average, using MO_{DMA} compared to MO_D. Unlike the aggregated reductions reported in Table 9.7, we perform a per-benchmark comparison between MO_{DMA} and MO_D in Table 9.8. Due to the use of the DMA controller instead of the CPU to copy code dynamically during runtime, MO_{DMA} provides better solutions in terms of WCET and energy objectives than MO_D. The reduction in WCET and energy consumption depends on the number of memory objects dynamically allocated within each solution. Therefore, it is difficult to extrapolate a trend between the relative size of SPM and the reduction in objective values due to the use of the DMA controller. But, in general, if we assume that the bigger the size of SPM, the more memory objects can be allocated dynamically to SPM, and in that case, MO_{DMA} will provide a larger reduction in WCET and energy objectives. Moreover, if we consider benchmarks with most memory objects that cannot be dynamically allocated either due to their size or liveness conflicts, the

Table 9.9.: Total number of benchmarks with best (or nondominated) quality indicators for 20%, 40%, and 80% relative SPM size configuration

Relative SPM size (%)	Quality metrics	MO_S FPA	MO_S SPEA	MO_D FPA	MO_D SPEA	SO_D ILP	SO_{DMA} ILP	MO_{DMA} FPA	MO_{DMA} SPEA	BL
20%	$\mathcal{C}$	0 (2)	2 (2)	0 (0)	0 (0)	0 (0)	1 (0)	1 (8)	7 (7)	0 (0)
	NR	0 (0)	3 (0)	0 (0)	0 (0)	0 (0)	1 (0)	8 (2)	6 (2)	0 (0)
	RNS	0 (2)	2 (2)	0 (0)	0 (0)	0 (0)	1 (0)	1 (8)	7 (7)	0 (0)
	HV	0 (1)	2 (1)	0 (0)	0 (0)	0 (0)	1 (0)	0 (10)	6 (10)	0 (0)
40%	$\mathcal{C}$	0 (1)	2 (2)	0 (0)	0 (1)	1 (0)	0 (0)	4 (7)	4 (7)	0 (0)
	NR	0 (0)	2 (1)	0 (0)	0 (0)	0 (0)	0 (0)	10 (4)	4 (4)	0 (0)
	RNS	0 (1)	2 (2)	0 (0)	0 (1)	1 (0)	0 (0)	4 (7)	4 (7)	0 (0)
	HV	0 (0)	3 (0)	0 (0)	1 (0)	1 (0)	0 (0)	2 (10)	3 (10)	0 (0)
80%	$\mathcal{C}$	1 (4)	2 (2)	0 (0)	1 (0)	0 (1)	0 (1)	2 (10)	2 (9)	0 (0)
	NR	0 (3)	2 (2)	0 (0)	1 (0)	0 (0)	0 (0)	9 (3)	2 (2)	0 (0)
	RNS	1 (4)	2 (2)	0 (0)	1 (0)	0 (1)	0 (1)	2 (10)	2 (9)	0 (0)
	HV	4 (1)	2 (1)	0 (0)	1 (0)	0 (0)	1 (0)	1 (7)	2 (7)	0 (0)

reduction in WCET and energy objectives will be comparatively less. In either scenario, we can conclusively say that MO_{DMA} outperforms MO_D for any SPM size configuration.

Table 9.9 provides the total number of benchmarks with best or nondominated $\mathcal{C}$, NR, RNS, and HV metrics for each approach, using the 20%, 40%, and 80% relative SPM size configurations. Similar to Table 9.3, in Table 9.9, the numbers shown outside the bracket represent the total number of benchmarks with the best quality indicators, and the numbers inside the bracket represent the number of benchmarks with nondominated Pareto optimal solutions compared to other approaches.

MO_S–SPEA provided the best solutions in terms of NR for a total of 3, 2, and 2 benchmarks in 20%, 40%, and 80% relative SPM size cases, respectively, and two benchmarks with the best solutions in terms of $\mathcal{C}$, RNS, and HV metrics in all SPM size cases. MO_S–FPA performed best for only one benchmark in terms of $\mathcal{C}$ and RNS metrics, and for 4 benchmarks in terms of HV metric in the 80% relative SPM size case. For the multi-objective static allocation approach, MO_S–SPEA performed better than MO_S–FPA for 20% and 40% SPM size cases, whereas MO_S–FPA performed slightly better than MO_S–SPEA for the 80% SPM size case.

MO_D–SPEA was only able to provide nondominated solutions for one benchmark in terms of $\mathcal{C}$ and RNS in the 40% SPM size case. Moreover, MO_D–SPEA performed best for one benchmark in terms of the HV metric for the 40% SPM size case and for a single benchmark in terms of all quality metrics for the 80% SPM size case. MO_D–FPA contributed no solutions to the final PF_T for any benchmark in any SPM size configuration.

SO_{DMA} provided the best solution in terms of all quality metrics, only for one benchmark in the 20% SPM size case. Moreover, for one benchmark,

Table 9.10.: Comparison between the average number of solutions (in %) on the final PF_T from respective PF_F of each optimization approach for 20%, 40%, and 80% relative SPM size configuration

Relative	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
SPM size (%)	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
20%	1.59	6.58	0.59	1.22	0	5	63.57	70.33	0
40%	2.74	11.86	0	1.19	5	0	75	65.91	0
80%	34.21	29.73	0	1.32	5	5.26	72.76	66.23	0

SO_{DMA} provided a nondominated solution in terms of $\mathcal{C}$ and RNS, and the best solution in terms of HV metric in the 80% relative SPM size case. For one benchmark, SO_D contributed the best solution in terms of $\mathcal{C}$, RNS, and HV in the 40% relative SPM size case, and a nondominated solution in terms of $\mathcal{C}$ and RNS in the 80% SPM size case.

From the overall table, we can see that MO_{DMA} provided better quality solutions for most benchmarks on average compared to all other approaches. For the 20% relative SPM size, MO_{DMA}–SPEA provided better solutions in terms of $\mathcal{C}$, RNS, and HV, and MO_{DMA}–FPA was better in terms of NR, i.e., MO_{DMA}–FPA produced the largest number of solutions on the final Pareto front PF_T, whereas the ratio of solutions from individual Pareto fronts PF_Fs that ended up on PF_T was higher for MO_{DMA}–SPEA.

For the 40% SPM size configuration, MO_{DMA}–FPA performed better in terms of NR, and MO_{DMA}–SPEA performed slightly better in terms of the HV metric. Whereas, MO_{DMA}–FPA and MO_{DMA}–SPEA both performed equally well in terms of $\mathcal{C}$ and RNS metrics. For the 80% SPM size configuration, MO_{DMA}–FPA performed better in terms of $\mathcal{C}$, NR, and RNS, and MO_{DMA}–SPEA performed slightly better than MO_{DMA}–FPA in terms of the HV metric.

Table 9.10 provides the percentage of the number of solutions on the final PF_T, on average, out of the solutions on the PF_F of each optimization run for 20%, 40%, and 80% SPM sizes relative to the benchmark, respectively. From the overall evaluation, we can see that for any SPM size, the proposed MO_{DMA} optimization provided the largest number of solutions on the final Pareto front. For the 20% SPM size, MO_{DMA}–SPEA produced the most solutions on the final Pareto front. For the remaining SPM configurations, MO_{DMA}–FPA had the largest number of solutions on the final Pareto front.

In the case of MO_S, we can see that the percentage of solutions contributed to the final Pareto front increases with an increase in the SPM size. MO_S–SPEA contributed more solutions on the final Pareto front than MO_S–FPA for all SPM size configurations. Finally, MO_D, SO_D, and SO_{DMA} contributed some solutions for a few benchmarks on the final Pareto front.

The number of basic blocks that can fit in SPM increases with an increase in SPM size. Therefore, in the case of static SPM allocation-based optimization (MO_S), the probability of better quality solutions also increases with an

increase in SPM size—evident from Table 9.10. However, we cannot draw such conclusions regarding MO_{DMA} trends. If a benchmark consists of functions and loops with small sizes and no liveness conflicts, a 20% SPM size configuration can also yield a great reduction in WCET and energy objectives. On the other hand, if a benchmark consists of functions and loops that barely fit in the SPM and have conflicting live ranges, even with an 80% SPM size configuration, the benchmark might not have good optimization potential. Therefore, based on the underlying benchmark and Equation 8.10, a system designer should decide which optimization, i.e., MO_{DMA} or MO_S, has more minimization potential. But, we can clearly state that MO_{DMA} will always perform better than MO_D, SO_D, SO_{DMA}, and BL.

9.6 Conclusion

In this chapter, we proposed a DMA cost model and a DMA analyzer within the WCC compiler. We also proposed a DMA Call Placement Optimization (DCPO), which optimizes the total execution time required for DMA transfer through the strategic placement of DMA calls within the code. We solved the DCPO problem using FPA, SPEA, and ILPs and conducted a comparative evaluation of the quality of DMA call placement achieved by each method. DCPO–ILP provided the most optimal solutions for all benchmarks compared to the FPA- and SPEA-based approach. Therefore, the ILP-based method is the best approach to solve the DCPO problem.

We also proposed a DMA-aware dynamic SPM allocation-based multi-objective optimization (MO_{DMA}) that dynamically allocates memory objects from the Flash memory to SPM using the DMA controller during runtime, aiming to minimize the WCET and energy consumption of the program. Due to the multi-objective nature of the MO_{DMA} optimization problem, we solved MO_{DMA} using two metaheuristic algorithms—FPA and SPEA.

In our evaluations, we used DCPO–FPA and DCPO–ILP while solving the MO_{DMA} problem and compared the proposed approach with previously proposed techniques. We evaluated the performance of MO_{DMA} by comparing the quality of its solutions with those obtained using MO_S, MO_D, SO_D, SO_{DMA}, and BL. MO_{DMA} clearly outperformed the BL evaluation run. The single-objective optimizations (SO_D and SO_{DMA}) failed to compete with MO_{DMA}. Due to the large WCET and energy overheads required for CPU-based dynamic allocation, MO_{DMA} outperformed MO_D. Moreover, due to the granularity difference of the basic block- and memory object-level allocation, static allocation-based multi-objective optimization MO_S occasionally produces better solutions. However, on average, MO_{DMA} generated a greater number and variation of final Pareto optimal solutions for most benchmarks.

The performance of memory allocation techniques depends on the size of the memory to which the allocation occurs. Therefore, we evaluated all the approaches for four different SPM sizes—20%, 40%, 60%, and 80%—relative

to the benchmark size and compared the solution quality obtained from them. The proposed MO_{DMA} optimization outperformed MO_D, SO_D, SO_{DMA}, and BL for all SPM size configurations. The percentage of solutions, on average, provided on the final Pareto front by MO_S increased with an increase in relative SPM size, as the static allocation-based technique can allocate more basic blocks to SPM as SPM size increases. However, on average, MO_{DMA} performed better than other approaches for all SPM size configurations.

Dynamic allocation techniques require jump corrections and extra code to facilitate dynamic allocation, increasing the code size. In Chapter 8, we observed that the MO_D optimization caused, on average, a 52.76% increase in code size compared to the original BL benchmark size. Similarly, as mentioned in Subsection 9.5.3.1, during evaluations for MO_{DMA}+DCPO–FPA, we observed an average increase of 45.92% in code size compared to the original BL benchmark size. Using a DMA controller, we aimed to reduce the WCET and energy overhead required by the CPU during dynamic allocation. However, MO_{DMA} does not consider code size as an objective while solving the optimization problem. Therefore, in the next chapter, we integrate code compression into a multi-objective dynamic allocation technique to decrease the code size of the dynamically allocatable code. The compression-aware multi-objective optimization proposed in the next chapter aims to minimize the program's WCET, energy consumption, and code size simultaneously.

10

Compression-aware Multi-Objective Dynamic SPM Allocation

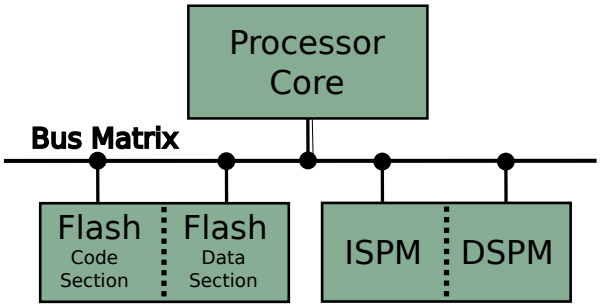
In Chapter 8, we proposed a compiler-level dynamic SPM allocation-based multi-objective optimization. However, we observed a 52.76%, on average, code size increase due to the code transformations required to generate a dynamically allocatable code. Similarly, the DMA-aware multi-objective dynamic SPM allocation proposed in the previous chapter led to an increase in code size. In Chapter 9, we observed an average 45.92% increase in code size due to the DMA-aware dynamic SPM allocation-based code transformations. Moreover, we need to add `memcpy()`- or DMA transfer-related functions to the code, further contributing to the code size.

Modern real-time systems—that need to react within a certain deadline to avoid failure—can be small, mobile, and battery-operated. Moreover, the increasingly complex embedded applications and limited memory space can restrict the permissible code size. Therefore, we consider the WCET, energy consumption, and code size as key optimization properties in this chapter.

By exploiting SPM dynamically, we optimize the program's WCET and energy consumption without being constrained by a static SPM size limit, contrary to static SPM allocation techniques. However, we need to perform certain code transformations to facilitate the jumping of code from the Flash to SPM and add function calls to initiate dynamic allocation of memory objects during runtime. Moreover, such code transformations can require spilling registers, adjusting the stack, fixing unreachable load/store instructions, etc., which increases code size. Therefore, to control such code size increase and reap the benefits of dynamic allocation, we propose a compression-aware multi-objective dynamic SPM allocation optimization (hereafter referred to as MO_C) that aims to optimize WCET, energy consumption, and code size simultaneously.

The optimization objectives in a multi-objective optimization can contradict each other. For example, minimizing WCET could increase energy consumption and vice versa, whereas dynamically allocating objects to SPM decreases WCET and energy but increases code size. To manage these trade-offs and solve the MO_C problem, we use two metaheuristic algorithms—FPA and SPEA. After solving the MO_C optimization problem using these metaheuristic

Figure 10.1.: Exemplary architecture for compression-aware multi-objective dynamic SPM allocation optimization



algorithms, we get Pareto fronts in a 3-dimensional objective space where no solution is better or worse than the others, representing the trade-offs between each objective.

Embedded systems need lightweight compression and decompression algorithms that do not impose high costs during runtime in terms of execution time and energy consumption. For the proposed MO_C approach, the memory objects are compressed at compile time and decompressed later during runtime into SPM and executed. Consequently, we do not need to consider the performance and energy costs due to the compile-time compression in the proposed MO_C optimization. However, as the decompression of memory objects takes place during runtime, its impact plays a crucial role in decisions made by the optimization, needing a lightweight decompression technique as it occurs during runtime. Thus, we use an asymmetric compression method with the help of the FastLZ library [Hid07] for compression and decompression. In asymmetric compression methods, the compression and decompression processes have different computational complexities or resource requirements. The FastLZ library employs resource-intensive compression during compilation, but the decompression of compressed objects during the execution of the program is simple and fast.

Similar to Chapter 8 and Chapter 9, in this chapter, we consider functions and loops as dynamic allocation candidates, referred to as memory objects. The optimization assumes an architecture with a Flash and an SPM, each with its respective code and data sections. Figure 10.1 shows an exemplary architecture for the compression-aware optimization used in this chapter. The underlying architecture consists of a Flash divided into code and data sections, an Instruction SPM (ISPM), and a Data SPM (DSPM). Unless explicitly specified, an SPM always refers to the ISPM in this chapter. Moreover, the dedicated Flash data section is hereafter referred to as a data section, and the Flash always refers to the Flash code section unless specified otherwise. During evaluations in Section 10.5, we compile and analyze code for the ARM Cortex-M0 microcontroller with the aforementioned memory hierarchy.

Figure 10.1 exactly represents the STM32F0xx implementation of the ARM Cortex-M0 architecture [STM22a].

The code generated after the optimization stores the compressed objects in the data section, and SPM acts as a fast buffer where we decompress objects during runtime. Well-placed decompression function calls within the code initiate the decompression of these objects during runtime.

After solving the MO_C problem, we expect solutions with minimized WCET and energy, and a code size less than the original. However, to perform dynamic allocation during runtime, we perform code transformations to enable dynamism, increasing the overall code size. Therefore, we do not have any code size constraint for the proposed MO_C optimization problem. Also, as MO_C has three equally important objectives, the optimization should be capable of finding such solutions if they exist, where WCET, energy consumption, and code size are minimized simultaneously. If MO_C does not find solutions with a code size less than the original, we still get a controlled increase in code size due to compression. The key takeaways from this chapter are:

- We compress memory objects at compile-time and dynamically decompress them into SPM at runtime.
- We use asymmetric compression that allows for fast and simple decompression at runtime.
- We estimate WCET and energy consumption of the compression-aware dynamically allocatable code as precisely as possible at compile-time.
- We propose a compression-aware multi-objective dynamic SPM allocation optimization (MO_C).
- MO_C aims to minimize WCET, energy consumption, and code size objectives simultaneously.

During evaluations, we compared the proposed MO_C optimization with the multi-objective dynamic allocation optimization (MO_D) approach from Chapter 8 and a baseline (BL) evaluation, where the whole program is executed from the Flash. We showed that MO_C surpasses them in terms of the quality of the obtained solutions. The key results from the evaluations are:

- MO_C–FPA and MO_C–SPEA, on average, provided 39.19% and 68.94% of solutions on the final Pareto front, respectively, compared to the MO_D and BL approaches.
- MO_C–SPEA provided the largest reduction of 37.46% and 13.21% in terms of WCET and energy consumption compared to BL, respectively.
- We observed only a 5.43% average increase in code size due to the MO_C optimization over BL.

- MO_C achieved a 25.48% average reduction in its own code size due to compression of memory objects.

This chapter is organized as follows: Section 10.1 provides an overview of the related background work. Section 10.2 defines and discusses compression-aware dynamic SPM allocation-related preliminaries and mathematical formulation for the MO_C problem. Section 10.3 proposes steps for generating compression-aware dynamically allocatable code. Section 10.4 presents the metaheuristic algorithm-based MO_C optimization. Section 10.5 presents evaluation results, and Section 10.6 concludes the chapter.

10.1 Related Work

Some compiler-level code compression techniques have been proposed in the past, but none have been explored in the context of multi-objective optimization. Code compression techniques can be further divided into hardware- and software-based techniques. Software-based compression/decompression techniques can be less expensive, flexible, and architecture-independent. Hardware-based approaches can provide better code compression, along with better performance and a reduction in energy consumption [WC92, XWL03, OHRC09]. However, they are more expensive and harder to implement without any hardware-level changes compared to software-based techniques. In this chapter, we use a software-based code compression/decompression technique that can be exploited within the WCC compiler without any hardware extensions.

Ros et al. [RS03] used a dictionary-based compression scheme to investigate its performance when applied to various compiler optimizations and parallel instruction orderings. A dictionary compression scheme can be used to determine the most frequent instructions and encode them with smaller-sized so-called codewords. These codewords are used as indices to locate the decompressed text streams from the dictionary table. Lefurgy et al. [LBCM97] investigated dictionary compression schemes for fixed- and variable-length codewords. Moreover, Lefurgy et al. [LPM00] proposed a fast decompression technique using software-managed caches as a decompression buffer.

Ozturk et al. [OKD+04] proposed an approach to compress data objects to free up space in SPM. If a data object is assigned to the SPM, the object currently occupying the SPM space is spilled to the main memory or compressed and stored back in the SPM. Ozturk et al. [OCKK06] further proposed an automated compiler support for data compression and decompression.

Pinter et al. [PW07] proposed a basic block-level code compression technique to reduce the storage space of a program, which could lead to a reduction of access time to off-chip memory. They proposed a code region selection scheme that might reduce the code size significantly after compression and

incur a reasonable runtime overhead. The authors used profiling information to select the code regions and direct code compression.

Muts et al. [MLF19] proposed a single-objective compression technique that tries to minimize the weighted sum of the WCET and the code size of a program (SO_F). SO_F is an ILP-based optimization technique that considers only functions as compression candidates and places compressed functions in the data section at compile-time. The SO_F optimization treats SPM as a buffer and statically decompresses the compressed functions into SPM during startup. The SO_F optimization comprises a selector, which selects functions as potential compression candidates if the functions have compression ratios (CR) (cf. Equation 10.5) less than 1, i.e., the functions with $CR > 1$ were not considered as compression candidates during the SO_F optimization. Unlike the proposed MO_C approach, SO_F has a code size constraint where the optimal solution's code size cannot be more than the original (BL) code size. Moreover, SO_F also lacks the dynamism of MO_C.

Some previously proposed approaches deal with dynamic allocation, some with minimizing WCET or runtime performance, and some focus on code compression, but none address WCET, energy consumption, and code size objectives simultaneously.

10.2 Dynamic SPM Allocation

Dynamic SPM allocation is a compiler-level optimization that dynamically allocates objects from the slow Flash memory to fast and energy-efficient SPM during runtime. This section is divided into the following subsections: Subsection 10.2.1 presents some dynamic SPM allocation preliminaries and defines so-called compression objects. It also presents the mathematical formulation for the MO_C optimization problem. Subsection 10.2.2 provides an overview of the code transformations required to generate the compression-aware dynamically allocatable code.

10.2.1 Problem Formulation

In this chapter, we consider the compression-aware dynamic allocation of memory objects ($\mathcal{M}_{(p,m)}$)—functions and loops—during runtime and static allocation of global data objects ($\mathcal{G}_g$) at compile time.

Let $x = (x_{(1,1)}, .., x_{(1,M_1)}, x_{(2,2)}, .., x_{(P,M_P)}, x_{(\tilde{d}+1)}, .., x_d) \in X$ be a d-dimensional binary decision variable vector, where $X \in \{0,1\}^d$ is the search space of the MO_C problem, and $\tilde{d} = \sum_{p=1}^{P} M_p$ is the total number of memory objects within the code. Each variable in subvector $x_{(1,1):(P,M_P)}$ decides if memory objects will be compressed at compile time and dynamically decompressed in the ISPM during runtime or not. By construction, $x_{(p,1)}$ is the decision variable for the p^{th} function f_p itself, and $x_{(p,2):(p,M_p)}$ are decisions associated with the

loops within the function f_p. Each variable in subvector $x_{(\tilde{d}+1):d}$ decides if the global data variable will be statically allocated to DSPM or not.

Similar to Chapters 8 and 9, we apply the three constraints discussed in Section 8.2.2 to the binary decision variable vector x while solving the compression-aware multi-objective dynamic SPM allocation problem in this chapter.

1. As we need at least one function in the Flash to call the decompression routines during runtime, the first constraint (Equation 8.3) is applied to the *entry point* function and its entry basic block if the entry basic block is part of a loop memory object.

2. The second constraint (Equation 8.4) is applied to the loops within a function if the function is a compression candidate according to the decision variable vector or solution vector x.

3. As global data variables ($\mathcal{G}_g$) are allocated statically to DSPM according to the decision subvector $x_{(\tilde{d}+1):d}$, the third constraint (Equation 8.5)—a DSPM size constraint—is applied to the global data objects.

Two memory objects simultaneously live, i.e., in use or being executed, should not occupy the same address range in SPM during runtime. Therefore, we perform liveness analysis on the program to determine the live range of memory objects. Function-level liveness analysis is discussed in detail in Subsection 8.3.2 in Chapter 8.

As explained in Chapters 8 and 9, memory objects might not be in consecutive addresses in the Flash memory, so we dynamically allocate address objects instead of memory objects. For example, a nested loop with 6 basic blocks could have the first and the last 3 basic blocks separated by another loop in the Flash. In such cases, we cannot decompress the whole memory object using one decompression function call. Therefore, similar to address objects, in this chapter, we define a compression object that is decompressed during runtime into SPM as follows:

Definition 10.1 (Compression Object)
A compression object $\mathbb{CO}_{(p,m,c)}$ is defined as a set of basic blocks from a memory object $\mathcal{M}_{(p,m)}$ that are in consecutive memory addresses within the Flash memory that can be compressed such that during runtime it can be decompressed into SPM using one decompression function call, i.e.,

$$\begin{aligned}\mathbb{CO}_{(p,m,c)} =\{b_p^r \in \mathcal{M}_{(p,m)} \mid ((b_p^{(r+1)} \in \mathcal{M}_{(p,m)} \implies \alpha^r + \beta^r = \alpha^{(r+1)}) \vee \\ (b_p^{(r-1)} \in \mathcal{M}_{(p,m)} \implies \alpha^r = \alpha^{(r-1)} + \beta^{(r-1)})) \vee \left|\mathcal{M}_{(p,m)}\right| = 1\},\end{aligned} \tag{10.1}$$

where $c = \overline{1, C_{(p,m)}}$, $C_{(p,m)}$ *is the total number of compression objects associated with* $\mathcal{M}_{(p,m)}$, b_p^r *is the* r^{th} *basic block of function* f_p, α_p^r *is the start address of* b_p^r *in the Flash, and* β_p^r *is the size of* b_p^r.

To dynamically decompress $\mathbb{CO}_{(p,m,c)}$ during runtime and to determine the ISPM address for each $\mathbb{CO}_{(p,m,c)}$, we need to solve the address assignment problem. The address assignment algorithm uses first- and best-fit heuristics to determine the ISPM address ranges for each $\mathbb{CO}_{(p,m,c)}$ such that no two compression objects with conflicting live ranges occupy the same addresses in the ISPM. The first-fit heuristic tries to fit as many $\mathbb{CO}_{(p,m,c)}$ as possible within the ISPM until it is full. If the ISPM is full or does not have enough free capacity to accommodate the decompression of the current considered $\mathbb{CO}_{(p,m,c)}$, we use the best-fit heuristic instead to find the best place in ISPM for its decompression. The address assignment algorithm for address objects is explained in detail in Subsection 8.3.3 in Chapter 8. We cannot assign overlapping ISPM addresses to two compression objects associated with the same $\mathcal{M}_{(p,m)}$. Therefore, the best-fit heuristic enforces the following liveness conflict constraint:

$$\begin{aligned} &\gamma_{(p,m,c)} + \beta_{(p,m,c)} \leqslant \gamma_{(p,m,c')} \vee \\ &\gamma_{(p,m,c')} + \beta_{(p,m,c')} \leqslant \gamma_{(p,m,c)} \; \forall \, c \neq c' \end{aligned} \tag{10.2}$$

where $\gamma_{(p,m,c)}$ and $\gamma_{(p,m,c')}$ are ISPM start addresses assigned to $\mathbb{CO}_{(p,m,c)}$ and $\mathbb{CO}_{(p,m,c')}$. Moreover, compression objects with overlapping executions, i.e., with live range conflicts, are also constrained by the best-fit heuristic, such that they will be decompressed in different ISPM address ranges during runtime.

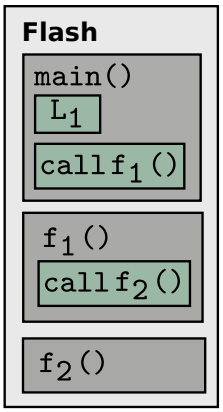
$$\begin{aligned} &\gamma_{(p,m,c)} + \beta_{(p,m,c)} \leqslant \gamma_{(p',m',c')} \vee \\ &\gamma_{(p',m',c')} + \beta_{(p',m',c')} \leqslant \gamma_{(p,m,c)} \quad \text{if } ((p',m') \neq (p,m)) \wedge \\ &\qquad (\mathbb{CO}_{(p,m,c)} \cap (\mathbb{CO}_{(p',m',c')} \cup \mathcal{L}_{(p',m')}) \neq \emptyset) \end{aligned} \tag{10.3}$$

The term $(\mathbb{CO}_{(p,m,c)} \cap (\mathbb{CO}_{(p',m',c')} \cup \mathcal{L}_{(p',m')}))$ implies that if $\mathbb{CO}_{(p,m,c)}$ and the live range of $\mathbb{CO}_{(p',m',c')}$ have any common basic blocks, then the resulting set will be non-empty, and the constraint in Equation 10.3 will be applied.

The compression-aware multi-objective dynamic SPM allocation (MO_C) problem proposed in this chapter is a 3-dimensional optimization problem. MO_C aims to simultaneously minimize the program's WCET, energy consumption, and code size. The following is the mathematical formulation for the MO_C optimization problem.

$$\begin{aligned} \min_{x \in X} \quad & F(x) = (F_1(x), F_2(x), F_3(x)) \\ \text{subject to} \quad & \text{Eq. 8.3, Eq. 8.4, Eq. 8.5, and} (\mathbf{C} - \epsilon) = 0 \end{aligned} \tag{10.4}$$

Figure 10.2.: Exemplary original code before compression. We assume that the loop L_1, the function `f1()`, and the function `f2()` will be dynamically decompressed during runtime.



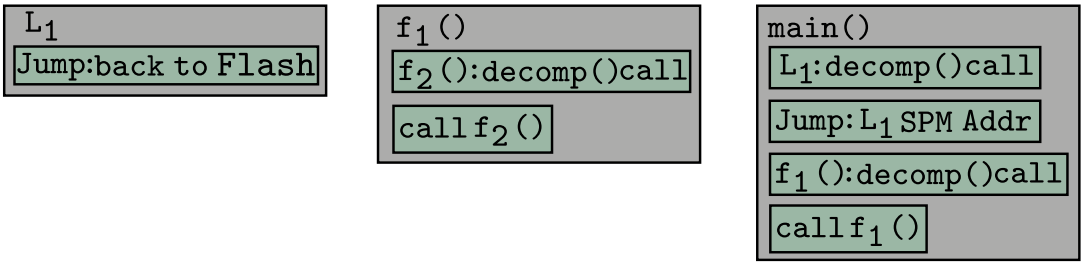
where $F_1(x) \in \mathbb{R}$ is the WCET objective of the problem, $F_2(x) \in \mathbb{R}$ is the energy consumption objective, and $F_3(x)$ represents the code size objective for the solution vector $x \in X$. Equations 8.3 and 8.4 are constraints on the solution subvector $x_{(1,1):(P,M_P)}$ associated with memory objects for dynamic compression. Equation 8.5 is a DSPM size constraint on the solution subvector $x_{(\tilde{d}+1):d}$ associated with global data variables. $\mathbf{C}$ is the total number of $\mathbb{CO}_{(p,m,c)}$, and ϵ is the total number of $\mathbb{CO}_{(p,m,c)}$ assigned to SPM after solving the address assignment problem. The fourth constraint $(\mathbf{C} - \epsilon) = 0$ implies that all the compression objects must have assigned SPM addresses after solving the address assignment problem.

10.2.2 Contextual Overview

This subsection presents exemplary code transformations required to generate compression-aware dynamically allocatable code. To perform compression and decompression of the code, the WCC compiler supports the FastLZ library [Hid07]. The FastLZ library is a lossless data compression library used for compression and decompression of the code. It is implemented in portable C and integrated, compiled, and analyzed within the WCC compiler. It can perfectly reconstruct the original data from the compressed data, which is necessary in our case. FastLZ decompression is very simple and quite fast, which can help us minimize the WCET and energy costs incurred due to the decompression of $\mathbb{CO}_{(p,m,c)}$ during runtime.

Figure 10.2 provides a code example that we will use to provide an overview of various stages in the generation and execution of compression-aware dynamically allocatable code. In this example, the original program consists of one loop L_1 and three functions—`main()`, `f1()`, and `f2()`. L_1 is in the `main()` function, `f1()` calls function `f2()`, and `main()` calls function `f1()`. The

Figure 10.3.: Exemplary code transformations needed before generating dynamically allocatable code for the example from Figure 10.2



original code is placed in the Flash, and we assume that memory objects L_1, $f_1()$, and $f_2()$ are dynamic decompression candidates—`main()` is the *entry point* function, and by definition, it will remain in the Flash memory (cf. Equation 8.3). We assume that all memory objects are equivalent to compression objects in this example.

If compression objects are decompressed and executed from ISPM, the previously valid jump between a loop and its predecessor and successor basic blocks, and the call instructions for functions can become invalid. Figure 10.3 shows the required code transformations to enable the eventual runtime dynamic decompression of the memory objects L_1, $f_1()$, and $f_2()$. The jumps between L_1 in SPM and its predecessor and successor basic blocks in `main()` in the Flash might become invalid or unreachable during runtime. Therefore, we perform jump correction such that previously valid jumps to compression objects are replaced by new jumps to their respective runtime addresses in ISPM. In Figure 10.3, the jump correction code is added before L_1 in the `main()` function, and if necessary, a jump back to the Flash memory is added at the end of L_1. The call instructions for $f_1()$ and $f_2()$ are also fixed by jump correction if they are out of reach.

To decompress compression objects during runtime, we insert the FastLZ decompression function (`decomp()`) (cf. Figure 10.4) and its calls within the code such that the decompression of compression objects occurs at the correct ISPM address and in the correct order. We do not add the `decomp()` function calls inside a loop to avoid decompression of compression objects at every iteration. Therefore, we can compress all loop compression objects without any specific order. However, to compress a function, we must check if any other function compression object is called from it. For example, in Figure 10.2, we cannot compress function $f_1()$ before $f_2()$. As $f_2()$ is called from $f_1()$, we must first compress $f_2()$, insert $f_2()$'s decompression call information in $f_1()$, and then compress $f_1()$. Therefore, contrary to loop compression, we need to follow a comprehensive order while performing compression of functions. Section 10.3.3 proposes an algorithm that provides us with a comprehensive order for function compression.

Figure 10.4.: Exemplary compression-aware code at compile time

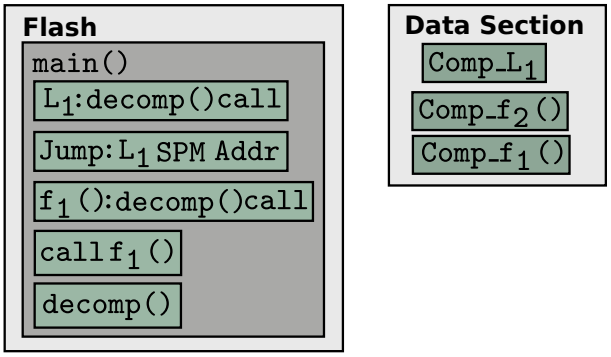
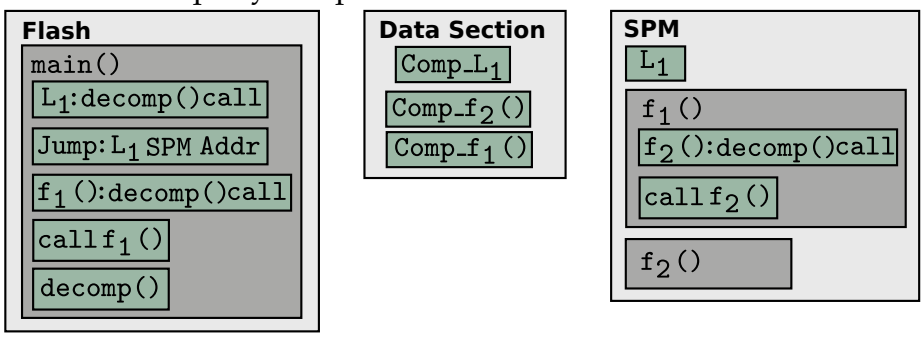


Figure 10.5.: Exemplary compression-aware code at runtime



The C file containing the FastLZ `decomp()` function is added to the code during the pre-processor phase by the WCC compiler. The compiler compiles `decomp()`, places it in the Flash memory, and makes it available for our use at the assembly level. At the assembly level, we insert the `decomp()` function calls at appropriate places. In Figure 10.3, we insert calls for the `decomp()` function in the `main()` function for L_1 that decompresses it and jumps to the SPM address of decompressed L_1. Similarly, we place FastLZ decompression calls for functions $f_1()$ and $f_2()$.

After all code transformations, we compress the compression objects in proper order and store them in the data section at compile time. We need to follow a certain order while compressing compression objects, which is derived using Algorithms 10.1 and 10.2. During runtime, the compressed objects are decompressed from the data section to ISPM and executed from ISPM. Figure 10.4 shows the exemplary compression-aware dynamically allocatable code at compile time. In the figure, we can see that all three compression objects—L_1, $f_1()$, and $f_2()$—are compressed into data objects named `Comp_`L_1, `Comp_`$f_1()$, and `Comp_`$f_2()$. These three data objects are stored in the data section, and the original loop and the functions from the Flash memory are deleted, reducing the size of the original code.

Figure 10.5 shows the exemplary compression-aware dynamically allocatable code at runtime. The compressed compression objects from the data section

are decompressed dynamically in SPM during runtime. In Figure 10.4, the decompression call for L_1 decompresses $Comp\_L_1$ from the data section to SPM, then the code jumps from Flash to SPM, executes L_1, and jumps back to Flash. Then, the decompression call for function $f_1()$ decompresses $Comp\_f_1()$ from the data section to SPM, and then the decompression call for $f_2()$ in function $f_1()$ decompresses $Comp\_f_2()$ from the data section to SPM during the execution of $f_1()$.

As loops are live only within the function, the compression and decompression of loops is straightforward. To compress loops at compile time, we extract the bytes corresponding to the loop from the program executable using its start address in the Flash (α) and its size (β). We know α and β for all compression objects ($\mathbb{CO}_{(p,m,c)}$) after solving the address assignment problem (cf. Algorithm 8.2 and Equations 10.2 and 10.3). We pass the extracted bytes to the FastLZ compression algorithm as input data, and the compressed data is returned to the WCC compiler. The compressed bytes are stored as data objects in the data section and used by the `decomp()` call during runtime. As compression is done at compile time, we do not include any compression routine in the final executable of the program.

To decompress $\mathbb{CO}_{(p,m,c)}$ during runtime, we need to provide a `decomp()` call with the following parameters: the size of the compressed $\mathbb{CO}_{(p,m,c)}$, its assigned SPM address, the original size of $\mathbb{CO}_{(p,m,c)}$, and the name of the compressed $\mathbb{CO}_{(p,m,c)}$ stored in the data section. In Figure 10.4, the parameters for the `decomp()` call for $f_1()$ are: the size of $Comp\_f_1()$, the SPM address γ for $f_1()$, the size of $f_1()$, and the name $Comp\_f_1()$.

10.3 Dynamic Compression

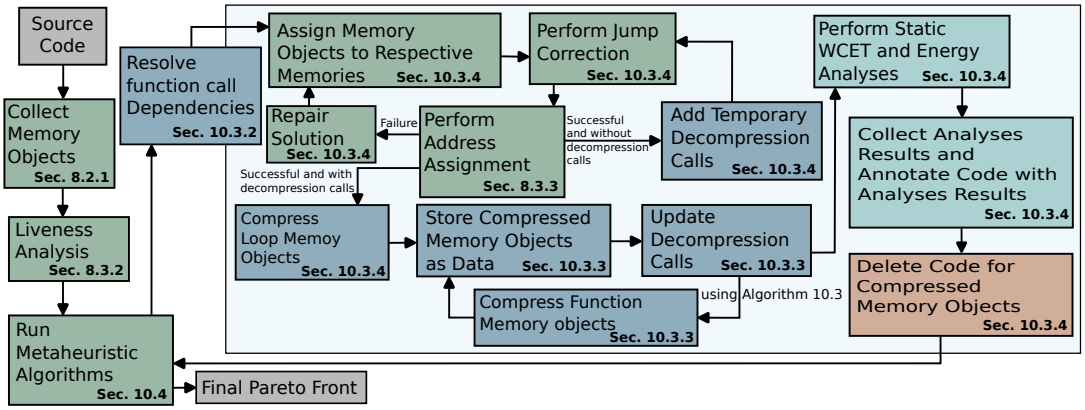
In this section, we discuss the steps required to generate compression-aware dynamically allocatable code. This section is divided into the following subsections: Subsection 10.3.1 provides the context and the steps required for compression-aware multi-objective dynamic SPM allocation and its overall picture within the compiler. Subsection 10.3.2 proposes an algorithm to resolve function call dependencies. Subsection 10.3.3 presents an algorithm to generate a particular order in which functions must be compressed at compile time. Subsection 10.3.4 presents the steps required for code generation and analysis of compression-aware dynamically allocatable code.

10.3.1 Contextual Overview

Figure 10.6 shows all the steps required to perform compiler-level MO_C. The steps for solving MO_C are proposed and described across Sections 10.2, 10.3, and 10.4 in this chapter and Section 8.3 in Chapter 8, and the sections associated with these steps are indicated within the figure.

To generate the compression-aware code, we need to identify and collect memory objects from the source code and perform liveness analysis to de-

Figure 10.6.: Compression-aware multi-objective dynamic SPM allocation framework



termine the live ranges of memory objects. Then, we call a metaheuristic algorithm (FPA or SPEA) to solve the multi-objective optimization problem (cf. Section 10.4). The metaheuristic algorithm generates binary solution vectors x that indicate which memory object will be dynamically decompressed during runtime.

For each x, we resolve function call dependencies to enable the correct decompression of functions during runtime (cf. Section 10.3.2). Then, we assign memory objects to their respective memories and perform jump corrections to accommodate jumps after decompression during runtime. We solve the address assignment problem to find suitable memory addresses for compression objects such that no liveness conflicts occur. If any compression object is not assigned an SPM address, the solution vector is repaired (cf. Section 10.3.4).

After address assignment, we add temporary decompression function calls, perform jump correction, and solve the address assignment problem again to accommodate decompression function calls (cf. Section 10.3.4). We compress loop compression objects—we do not need to follow any particular order here. We follow a comprehensive order for compressing function compression objects (cf. Section 10.3.3). Then, we update the decompression call parameters with the compressed compression object's information (cf. Section 10.3.3). Finally, we perform WCET, energy, and code size analyses, collect and annotate the code with analysis results, and delete the code for the compressed memory objects. The analysis results are used as objective values while solving the multi-objective optimization using the metaheuristic algorithms (cf. Section 10.4).

10.3.2 Resolving Function Call Dependencies

In this subsection, we propose an algorithm that aims to resolve dependencies between functions and the function calls within them. We need to resolve such

Algorithm 10.1 Resolving function call dependencies

```
1: Input: Solution vector x
2: Output: 𝔽
3: Initialize: 𝔽    ▷ Map of function and list of functions that it is dependent on
4: for p = 1 : P do
5:     for m = 1 : M_p do
6:         if x_(p,m) == 1 & m == 1 then
7:             Initialize: 𝒞ℱ_p    ▷ List of functions that f_p is dependent on
8:             for each basic block b_p^r do
9:                 if b_p^r calls f_p' & p' ≠ p & x_(p',1) == 1 then
10:                    𝒞ℱ_p ← f_p'    ▷ f_p is dependent on f_p'
11:                    Recursively add functions called by f_p' to 𝒞ℱ_p
12:            𝔽 ← {f_p, 𝒞ℱ_p}
13: for each function f_p in 𝔽 do
14:     for each function f_p' in 𝒞ℱ_p do
15:         for each function f_p'' in 𝔽 do
16:             if f_p' == f_p'' then
17:                 for each function f_p''' in 𝒞ℱ_p'' do
18:                     if f_p == f_p''' then    ▷ f_p depends on f_p' and vice versa
19:                         Compress f_p and f_p' and calculate CR_(p,1) and
                            CR_(p',1) for f_p and f_p'
20:                         if CR_(p,1) < CR_(p',1) then
21:                             x_(p,1) = 0
22:                         else if CR_(p,1) > CR_(p',1) then
23:                             x_(p',1) = 0
24:                         else
25:                             if size(f_p) ⩽ size(f_p') then
26:                                 x_(p,1) = 0
27:                             else
28:                                 x_(p',1) = 0
29:                         Update 𝒞ℱ_p or 𝔽 or both accordingly
```

dependencies to tackle function call conflicts and generate valid compression-aware dynamically allocatable code.

Algorithm 10.1 resolves function call dependencies between functions and builds a map $\mathbb{F}$ between a function and the list of functions that it is dependent on. For example, let us assume a function f_1 calls another function f_2, and f_2 needs to be compressed. As we need to add compressed f_2's information to its `decomp()` call in f_1, we cannot compress f_1 before compressing f_2. Therefore, the function f_1 is dependent on f_2 in this case.

Algorithm 10.1 also updates the solution vector x if there exist q functions forming a cycle, which results in a cyclical conflict for adding decompression

call information in either of the functions. For example, two functions that call each other form a cycle of 2 functions. As Algorithm 10.1 needs only the function call information and as the solution vector x might get updated, calling Algorithm 10.1 is the first step for generating compression-aware dynamically allocatable code.

For the solution vector x, we initialize the map $\mathbb{F}$ and collect a list $\mathcal{CF}_p$ of functions on which f_p is dependent (Lines 3-12). If a function $f_{p'}$ is called by a basic block b_p^r in f_p, where $f_{p'}$ is also a compression candidate, i.e., $x_{(p',1)} == 1$, and $p \neq p'$, then $f_{p'}$ is added to the list $\mathcal{CF}_p$ (Lines 9 and 10). If f_p calls itself, i.e., f_p is a recursive function, we can be sure that the function is already decompressed in SPM, and we do not need to add a decompression call before the call for f_p in f_p. We recursively add function calls to $\mathcal{CF}_p$ if function $f_{p'}$ calls another function, and so on, to establish cyclical function call dependencies between more than two functions (Line 12).

Next, we iterate over the map $\mathbb{F}$ and respective $\mathcal{CF}_p$ for each function f_p, and if f_p depends on $f_{p'}$ and vice versa, we compress both f_p and $f_{p'}$ to calculate their compression ratio CR (Lines 13-19).

$$\texttt{CR} = \frac{size(\ \textit{compressed object}\ )}{size(\ \textit{original object}\ )} \tag{10.5}$$

If f_p depends on $f_{p'}$ and $f_{p'}$ depends on f_p, we cannot insert exact decompression information in both functions before compressing either of the functions. For example, if function f_1 calls function f_2, f_2 also calls f_1 during its execution, and both f_1 and f_2 are compression candidates, there exists a cyclical dependency between f_1 and f_2. If we compress f_1 first and add a `decomp()` call for f_1 in f_2 and then compress f_2, we cannot add the `decomp()` call for f_2 in f_1 without changing f_1's code, thereby invalidating the previously compressed function f_1.

Therefore, in such cases, we calculate CR, and the function with lower compression potential is kept in the Flash during runtime (Lines 20-23). If both functions have the same CR, we keep the smaller-sized function in the Flash, hoping that placing the larger function in SPM might lead to higher WCET and energy optimization potential eventually (Lines 25-28). As map $\mathbb{F}$ only consists of functions that are dynamic compression candidates, we need to update $\mathcal{CF}_p$ or $\mathbb{F}$ or both if the solution vector x is updated.

In Figure 10.2, we do not see a cyclical conflict. However, for example, if function `f2()` also calls function `f1()`, we should have a cyclical conflict between the two functions. In this case, Algorithm 10.1 will resolve the conflict by updating the respective solution vector x and placing either function `f1()` or `f2()` back to the Flash.

Algorithm 10.2 Compression of functions

1: **Input:** The map $\mathbb{F}$ from Algorithm 10.1
2: Use the map $\mathbb{F}$ to build the graph $\mathbb{G} := (\mathbb{V}, \mathbb{E})$ ▷ $\mathbb{G}$ maps a graph between functions and the functions called by it
3: Use the DFS algorithm to traverse the graph $\mathbb{G}$ and get the topologically ordered vector $\mathbb{S}$ of functions
4: **for** each function f_p in $\mathbb{S}$ **do**
5: Update decompression call information in function f_p
6: Compress function f_p and place it in the data section

10.3.3 Function Compression

As mentioned before, we do not need to follow any specific order while compressing loops. However, we need to perform compression for functions in a particular order to generate valid dynamically allocatable code.

Algorithm 10.2 describes the steps required to compress the functions in a comprehensive order. First, we build a function call graph $\mathbb{G} := (\mathbb{V}, \mathbb{E})$ to represent the calling relationships between functions and the functions called by those functions (Line 2). $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ is a directed call graph, where $\mathbb{V}$ is the set of nodes, each node $f_p \in \mathbb{V}$ corresponds to a function in the program, $\mathbb{E}$ is the set of directed edges, and each edge $(f_p, f_{p'}) \in \mathbb{E}$ represents a call from function f_p to function $f_{p'}$. The graph $\mathbb{G}$ is built using the map $\mathbb{F}$ from Algorithm 10.1, which implies that $\mathbb{G}$ has no cyclical edges. Therefore, $\mathbb{G}$ is a directed acyclic graph, which allows the linear ordering of the nodes.

To decide the order in which the function compression occurs, we use the standard Depth-First Search (DFS) [App04] algorithm within the compiler (Line 3). Let $\mathbb{S}$ be the output of DFS that gives us the ordered vector of the functions in which function compression must occur. As $\mathbb{G}$ is a directed acyclic graph, DFS will always yield a valid topological order of functions [CLRS22]. As $\mathbb{S}$ holds the ordered vector of functions, the functions with no or least number of function calls are compressed first, and the functions with the highest number of dependencies are compressed last (Lines 4-6).

If a function is a compression candidate, we know that no loops contained within the function are separate compression candidates (cf. Equation 8.4). The decompression call information for a function call, if any, is updated in the function f_p, and f_p is compressed and placed in the data section.

For the example described in Figure 10.2, Algorithm 10.2 will first compress `f2()`, place the `Comp_f2()` in the data section, update the decompression call information for `f2()` in the function `f1()`, perform similar steps for the function `f1()`, and finally update the decompression call information in the `main()` function.

Algorithm 10.3 Generation and analysis of the code for compression-aware dynamic SPM allocation

1: **Input:** Solution vector x
2: **Output:** Compression-aware dynamically allocated code and its analysis results
3: **do**
4: Call Algorithm 10.1 to get $\mathbb{F}$, and the solution vector x is updated if necessary
5: Allocate memory objects to SPM using the solution vector x and perform jump correction
6: Perform address assignment using Algorithm 8.2 and collect compression objects ($\mathbb{CO}_{(p,m,c)}$)
7: Insert decompression function calls and perform jump correction
8: Perform address assignment after code transformations using Algorithm 8.2
9: **if** $(\mathbf{C} - \epsilon) \neq 0$ **then**
10: Repair the solution vector x
11: **while** $(\mathbf{C} - \epsilon) \neq 0$
12: Compress loop $\mathbb{CO}_{(p,m,c)}$, place them in the data section, and update the decompression call information
13: Call Algorithm 10.2 and compress functions if necessary according to the solution vector x
14: Generate a temporary static version of the code by placing uncompressed $\mathbb{CO}_{(p,m,c)}$ in SPM according to the solution vector x
15: Perform WCET and energy analyses, and calculate the uncompressed and compressed code size
16: Delete all uncompressed $\mathbb{CO}_{(p,m,c)}$ that were temporarily placed in SPM

10.3.4 Code Generation and Analyses

Algorithm 10.3 describes the steps required to generate and analyze compression-aware dynamically allocatable code. It takes the solution vector x as an input and provides compressed and analyzed code as output (Lines 1 and 2). We call Algorithm 10.1 to generate the map $\mathbb{F}$ to establish a relation between a function and functions called by it, and we update the solution vector x if any function call dependency conflicts exist (Line 4). We allocate memory objects to SPM or Flash according to the updated solution vector x and perform jump correction to fix broken jumps (Line 5).

We solve the address assignment problem, get compression objects ($\mathbb{CO}_{(p,m,c)}$) for all memory objects, and collect all source addresses (α), destination addresses (γ), and uncompressed sizes (β) for all $\mathbb{CO}_{(p,m,c)}$ (Line 6). To solve the address assignment problem, we use Algorithm 8.2 from Chapter 8.

Instead of address objects, in this chapter, we use a set of compression objects as an input for the address assignment algorithm.

We insert decompression calls in the assembly code based on the $\mathbb{CO}_{(p,m,c)}$ information received after solving the address assignment problem and perform jump corrections to accommodate these new decompression calls (Line 7). But, at this stage, we do not know the size of the compressed $\mathbb{CO}_{(p,m,c)}$—we add the exact size value after compressing $\mathbb{CO}_{(p,m,c)}$. Again, we solve the address assignment problem using Algorithm 8.2 to get the updated α, γ, and β information after inserting decompression calls (Line 8). If the address assignment problem is not able to assign SPM addresses to all $\mathbb{CO}_{(p,m,c)}$, we repair the solution vector x by moving the memory objects associated with the unassigned $\mathbb{CO}_{(p,m,c)}$ to the Flash, i.e., by setting their binary decision variables to 0 (Lines 9 and 10).

As mentioned in Subsection 10.2.1, $\mathbf{C}$ is the total number of $\mathbb{CO}_{(p,m,c)}$, and ϵ is the total number of $\mathbb{CO}_{(p,m,c)}$ assigned to SPM after solving the address assignment problem. $(\mathbf{C} - \epsilon) \neq 0$ implies that one or more $\mathbb{CO}_{(p,m,c)}$s were not assigned SPM addresses. If the solution vector x is repaired, we need to generate the compression-aware code again for the updated solution vector x (Lines 3-11).

Next, we compress all loops $\mathbb{CO}_{(p,m,c)}$, place the compressed $\mathbb{CO}_{(p,m,c)}$ in the data section, and update the size of the compressed $\mathbb{CO}_{(p,m,c)}$ information in the respective decompression calls (Line 12). Algorithm 10.2 is called to compress functions in a specific order according to the ordered vector $\mathbb{S}$ (Line 13). We temporarily assume that all memory objects will fit in the SPM and generate a temporary static version of the code by temporarily placing the uncompressed $\mathbb{CO}_{(p,m,c)}$ in SPM according to the solution vector x (Line 14). The static version of the code is used to perform static WCET and energy analyses at the compiler level (Line 15). The static WCET and energy analysis tools cannot deal with the actual runtime dynamism of decompression of compression objects from the data section to SPM. They also fail to understand that the same SPM address γ could be occupied by different compression objects during a program's execution. Therefore, we analyze the temporary static version that consists of all uncompressed $\mathbb{CO}_{(p,m,c)}$ already placed in SPM, which captures the runtime behavior of code execution from SPM. The calls of the decompression function in the code capture the extra WCET and energy consumption incurred due to the actual decompression of $\mathbb{CO}_{(p,m,c)}$ during runtime.

As mentioned in Section 10.2.2, the code for the FastLZ library is added during the pre-processor phase of the WCC compiler. As the FastLZ library code is required to decompress the compressed object during runtime, it is a part of the final binary. Therefore, the code size of the solution vector x contains the FastLZ library code too. We distinguish between different versions of the code and their sizes during compile time and runtime as follows:

- At compile time, before solving the MO_C problem, the code within the Flash memory consists of the original code without any modifications. Let the size of this original code be CS^O.
- To generate the compression-aware code based on the solution vector x, we compress the compression objects and place them in the data section. Let the size of the compressed $\mathbb{CO}_{(p,m,c)}$s in the data section be CS^D.
- After MO_C generates code based on the solution vector x, the original code for the $\mathbb{CO}_{(p,m,c)}$s is deleted from the Flash's code section, as mentioned before. Therefore, at this stage, Flash's code section consists of code that was not considered as compression candidates by the solution vector x. Let the size of this code in the Flash be CS^F.
- During runtime, the compressed $\mathbb{CO}_{(p,m,c)}$s are decompressed in the SPM. The total size of the decompressed $\mathbb{CO}_{(p,m,c)}$s during runtime is equal to the sum of the original sizes of the compression objects. Let this total size of the decompressed $\mathbb{CO}_{(p,m,c)}$s that will be allocated to the SPM during runtime be CS^S.

We can easily calculate these different code sizes at various stages of code generation within the WCC compiler. After performing WCET and energy analyses, we calculate two code sizes—the compressed code size (CS^C) and the uncompressed code size (CS^{UC})—within the WCC compiler (Line 15).

Definition 10.2 (Compressed Code Size)
CS^C *is the code size of the compressed code, which is calculated by summing the size of the code in the Flash (*CS^F*) and the size of the compressed compression objects in the data section (*CS^D*).*

$$CS^C = CS^F + CS^D \tag{10.6}$$

Definition 10.3 (Uncompressed Code Size)
CS^{UC} *is the code size of the uncompressed code, which is calculated by summing the size of the code in the Flash (*CS^F*) and the size of the decompressed compression objects that will be allocated to the* ISPM *during runtime (*CS^S*).*

$$CS^{UC} = CS^F + CS^S \tag{10.7}$$

MO_C compresses the $\mathbb{CO}_{(p,m,c)}$s and places them in the data section during compile-time; the compressed $\mathbb{CO}_{(p,m,c)}$s are decompressed during runtime. Consequently, the total code size after decompression at runtime consists of three components: the code that is never compressed and remains in the Flash memory (CS^F), the compressed $\mathbb{CO}_{(p,m,c)}$s stored in the data section (CS^D), and the decompressed $\mathbb{CO}_{(p,m,c)}$s residing in the SPM during runtime (CS^S). As the MO_D code does not consist of any compressed $\mathbb{CO}_{(p,m,c)}$s in the data section, the total runtime code size of MO_C will be larger than MO_D's code

Algorithm 10.4 Compression-aware Multi-Objective Dynamic SPM Allocation (MO_C)

```
1: Recognize and collect memory objects
2: Perform Liveness Analysis
3: for p = 1 : P do                    ▷ Run algorithm for P different initial populations
4:     Randomly initialize the initial population of size N
5:     for n = 1 : N do
6:         Call Algorithm 10.3 to generate compression-aware code
7:     while Stopping criteria are not reached do
8:         Update Individuals using respective update operators
9:         for Each updated Individual do
10:            Call Algorithm 10.3 to generate compression-aware code
11:        Update the next generation using the selection operator
12:    Collect the Pareto optimal solution set
13: return Final Pareto optimal solution set (PF_F)
```

size at compile time. However, the MO_C aims to minimize the code size by compressing memory objects at compile-time. As a result, MO_C cannot minimize the runtime code size that includes decompressed $\mathbb{CO}_{(p,m,c)}$s in the SPM, i.e., $CS^F + CS^D + CS^S$. Therefore, only the Flash-resident components, CS^F and CS^D, are considered when computing the compressed code size CS^C at compile time. Consequently, the minimization objective $F_3(x)$ in the MO_C problem (cf. Equation 10.4) is CS^C, i.e., $F_3(x) = CS^C$.

Moreover, CS^{UC} only calculates the code size if no compression is applied to the compression objects at compile time. Consequently, we only consider CS^F and CS^S while calculating the uncompressed code size (CS^{UC}) at compile time. We use CS^{UC} only for comparison purposes during evaluations in Section 10.5. After collecting all the analysis results, we delete all the uncompressed code placed in the ISPM to get the compression-aware dynamically allocatable code (Line 16).

10.4 Compression-aware Multi-Objective Dynamic SPM Allocation

In this section, we discuss the proposed compression-aware multi-objective dynamic SPM allocation optimization (MO_C) that simultaneously minimizes the program's WCET, energy consumption, and code size. In Subsection 10.2.1, we discussed and provided the mathematical formulation of the MO_C optimization problem. We use a metaheuristic algorithm to solve the 3-dimensional MO_C problem. During evaluations, we use two different metaheuristic algorithms—FPA and SPEA—to evaluate MO_C.

Algorithm 10.4 describes the steps required to solve the MO_C optimization problem. We first collect all memory objects within the program and perform liveness analysis to determine their live ranges before running the metaheuristic algorithm (Lines 1 and 2). As the metaheuristic algorithm uses a random initial population and the initial population can affect the convergence of the algorithm, we run the metaheuristic algorithm for P different initial populations to reduce the effect of randomization (Lines 3 and 4). We generate compression-aware code for the randomly initialized population, which serves as an input for the iterative metaheuristic algorithm (Lines 5 and 6). During each generation of the population, each individual is updated within the population using update operators that are specific to a particular algorithm (Line 8) (cf. Section 4.4).

For each updated individual, we call Algorithm 10.3 to generate and analyze compression-aware code (Lines 9 and 10). Based on the objective values, the metaheuristic algorithm chooses the top-scoring solutions using the selection operator based on the concept of Pareto dominance (Line 11) (cf. Definition 4.1 and Section 4.4).

We specify two stopping criteria to finish the iterative metaheuristic algorithm: the maximum number of generations and the maximum number of generations for which the population remains the same. We update the individuals in the populations until at least one of the stopping criteria is reached. After meeting one of the two stopping criteria, we collect the Pareto optimal solution set for the p^{th} random initial Population (Line 12). We generate a final Pareto optimal solution set out of the union of the Pareto optimal solutions from each iteration of the initial Population (Line 13). As the multi-objective optimization problems considered in this thesis are quite complex and the functions to formulate the objective functions of the problem are unknown, the true Pareto optimal solutions to these problems are unknown. Therefore, when we talk about the Pareto optimal solutions, we are referencing the approximated final Pareto optimal solution set.

10.5 Evaluation

In this section, we compare the proposed compression-aware multi-objective dynamic SPM allocation (MO_C) with the MO_D optimization from Chapter 8 and a baseline (BL) evaluation. The BL evaluation is performed by placing the whole program in the Flash memory. We solve the multi-objective MO_C and MO_D optimizations using two metaheuristic algorithms—FPA and SPEA.

In Chapter 8, we observed that MO_D caused 52.76% average code size increase compared to BL. In this section, we compare MO_C against MO_D to demonstrate MO_C's efficacy in comparison to the base multi-objective dynamic SPM allocation. Moreover, we also demonstrated that MO_D outperformed SO_D in Chapter 8. Therefore, for brevity's sake, we skip the comparison of MO_C with SO_D in this section.

As mentioned in Section 10.1, SO_F [MLF19] is an ILP-based single-objective optimization that compresses functions and decompresses all of them statically in SPM before executing the entry point function. During these evaluations, we observed that SO_F failed to provide optimal solutions for any of the `EEMBC` benchmarks. As mentioned in Chapters 8 and 9, all `EEMBC` benchmarks considered during evaluations have a large function, "`t_run_test()`", which does not fit completely in the SPM. Consequently, SO_F cannot consider them as compression candidates. Moreover, the rest of the small functions from the `EEMBC` benchmarks were not deemed optimal by SO_F's selector to be placed in ISPM. Therefore, for all benchmarks, SO_F returned the original (BL) code as the optimal solution due to its strict pre-selection phase, static ISPM size constraint, and strict code size constraint. SO_F is not the best approach to optimize `EEMBC`-like benchmarks. Therefore, we need to examine the underlying code before choosing the appropriate compiler-level optimization for the program. As we already compared the proposed MO_C with the BL evaluation run, we do not make any further comparisons between the MO_C and the SO_F solutions during these evaluations for brevity's sake.

This section is further divided as follows: Subsection 10.5.1 describes the evaluation setup used to solve the optimization problems in this chapter. Subsection 10.5.2 presents the Pareto optimal solutions found after solving the MO_C, MO_D, and BL. Subsection 10.5.3 performs code size value comparisons between MO_C, MO_D, and BL. Subsection 10.5.4 compares the quality of MO_C, MO_D, and BL solutions using the quality indicators described in Subsection 4.3.3. Subsection 10.5.5 presents the compilation time results for MO_C and MO_D.

10.5.1 Evaluation Setup

Similar to previous chapters, the evaluations are conducted using the ARM Cortex-M0 fork of WCC. The WCET and energy analyses are performed using `aiT` [Abs24] and `EnergyAnalyzer` [Tea20], respectively, and code size is easily calculated internally within the WCC compiler. We use the FastLZ library [Hid07] to compress code during compilation and to decompress compressed code at runtime.

We evaluated 20 benchmarks from the benchmark suites offered by the `EEMBC` [PCLGO09]. We assume that all the benchmarks will fit within the Flash memory. The code is generated using the compiler's `-O2` optimization level. The ISPM and the DSPM sizes are individually adjusted to 60% relative to the benchmark size to increase the pressure on the optimization.

MO_C is a multi-objective optimization that aims to simultaneously minimize three objectives—WCET, energy, and code size. MO_D is a multi-objective optimization that minimizes two objectives—WCET and energy consumption. So, we calculate the code sizes of the final Pareto optimal solutions to get the respective code size values. For the BL evaluation, we place the code

in the Flash, generate the executable, and perform WCET, energy, and code size evaluations. The values of the hyperparameters for the FPA and SPEA algorithms are described in Chapter 5 in Tables 5.1 and 5.2, respectively. The results obtained during these evaluations are valid for the algorithm parameters described above.

10.5.2 Pareto Fronts

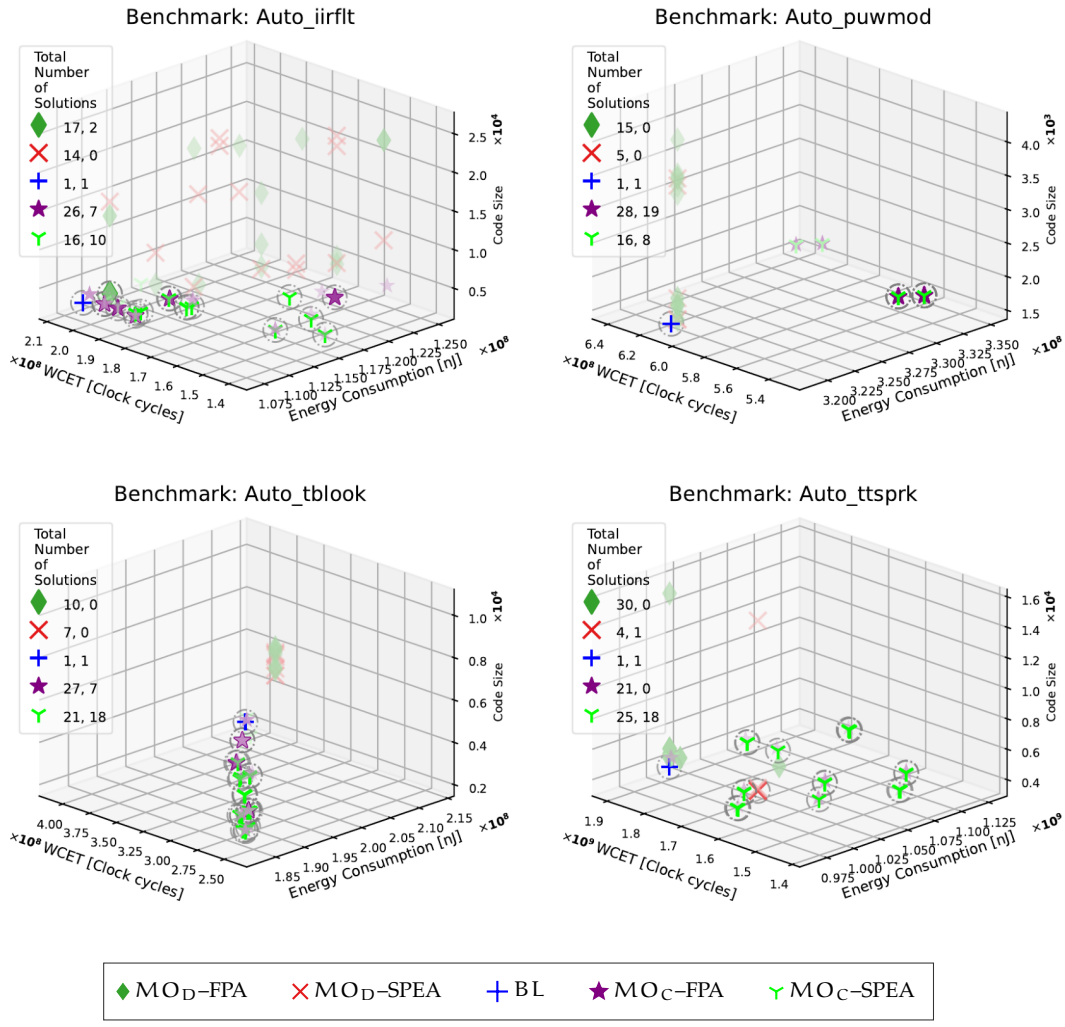
As obtaining the true Pareto front for a multi-objective optimization problem like MO_C is ambitious, similar to previous chapters, we realistically assume that the true Pareto front is unknown and construct the final Pareto front (PF_T) for each benchmark from the union of solutions obtained from all the optimization runs, i.e., $PF_T = \{p_t \mid \forall p_t \nexists q_j \in (PF_{F_C} \cup PF_{F_D} \cup PF_{F_{BL}}) \prec p_i\}$. p_t is the t^{th} non-dominated solution in PF_T; $PF_{F_C} = PF_{F_{MO_C-FPA}} \cup PF_{F_{MO_C-SPEA}}$ is the union of final Pareto optimal solutions obtained during the MO_C optimization runs using FPA ($PF_{F_{MO_C-FPA}}$) and SPEA ($PF_{F_{MO_C-SPEA}}$); similarly, PF_{F_D} is the union of final Pareto optimal solutions obtained during the MO_D optimization runs using FPA and SPEA; $PF_{F_{BL}}$ is the single solution obtained after running the BL evaluation. For brevity's sake, we refer to the final Pareto front obtained at the end of each optimization run only as PF_F.

In this subsection, we compared all the solutions obtained after performing MO_C, MO_D, and BL evaluation runs. Figure 10.7 shows the solutions found by MO_D–FPA, MO_D–SPEA, BL, MO_C–FPA, and MO_C–SPEA. For brevity's sake, we present the solutions for only four benchmarks. We present the figures for all remaining benchmarks in Appendix G.

Figure 10.7 presents 3D scatter plots for Pareto fronts obtained for `Auto_iirflt`, `Auto_puwmod`, `Auto_tblook`, and `Auto_ttsprk` benchmarks. The final PF_Ts are represented with darker colors, and the fainter version of those colors represents the PF_Fs of the respective optimization runs. For clarity's sake, the solutions on the final PF_T are highlighted by dotted gray circles. The left side legend of each plot shows the total number of solutions on PF_F for each approach and the number of solutions contributed by that approach to the final PF_T. The legend at the bottom of the figure presents the optimization run to which a particular solution belongs. The x-axis represents the WCET objective, the y-axis represents the energy consumption, and the z-axis represents the code size objective. For MO_D and BL, the code size objective on the z-axis represents the total size of the code generated at compile time. In the case of MO_C, the code size objective on the z-axis represents the compressed code size CS^C, as explained in Section 10.3.4.

For `Auto_iirflt`, MO_D–FPA, BL, MO_C–FPA, and MO_C–SPEA contributed solutions to the final PF_T. For `Auto_ttsprk`, MO_D–SPEA, BL, and MO_C–SPEA contributed solutions to PF_T. For both benchmarks, MO_C–SPEA contributed the largest number of solutions to the final PF_T.

Figure 10.7.: Pareto fronts for four benchmarks using MO_D, BL, and MO_C



In the case of `Auto_puwmod` and `Auto_tblook` benchmarks, MO_C–FPA, MO_C–SPEA, and BL contributed solutions to PF_T. For `Auto_puwmod`, MO_C–FPA contributed the most solutions, whereas MO_C–SPEA contributed the most solutions to the final PF_T for `Auto_tblook`.

Figures 10.7, G.1, and G.2 show that MO_D–FPA and MO_D–SPEA contribute solutions to PF_T for some benchmarks. In some cases, the solutions on the final PF_T provided by MO_D were better in terms of WCET or energy consumption (or both) than the solution provided by MO_C. However, none of these solutions were better in terms of code size, i.e., MO_C always provides solutions with smaller code sizes compared to MO_D solutions. It is evident from the evaluations in Subsection 10.5.3 as well.

In the case of all four benchmarks in Figure 10.7 and from some benchmarks in Figures G.1 and G.2, we can observe that the BL solutions also lie on the final PF_T. As discussed in Subsection 10.5.3, we observed an average increase

Table 10.1.: Comparison between the average number of solutions (in %) on the final PF_T from the respective PF_F of each optimization approach

MO_D		BL	MO_C	
FPA	SPEA		FPA	SPEA
5.29	10	50	32.97	67.23

Table 10.2.: Average reduction in WCET and energy consumption objectives obtained using MO_D and MO_C over the BL evaluation run

Objectives	MO_D		MO_C	
	FPA	SPEA	FPA	SPEA
WCET	36.7	36.74	27.57	37.46
Energy consumption	11.26	11.49	9.28	13.21

in MO_C's CS^C code size compared to the BL code size. Unlike SO_F, MO_C does not have a strict pre-selection phase: all functions and loops are part of the solution vector. We dynamically decompress the compression objects in ISPM: a strict ISPM size constraint does not apply to MO_C. As WCET, energy consumption, and code size are equal optimization objectives in MO_C, an increase in code size compared to the BL code size is allowed. Therefore, whenever the MO_C solutions have a larger code size than BL, the BL solution lies on the final PF_T.

For some benchmarks, MO_C showed an average increase in the code size compared to the BL (cf. Figure 10.8). However, this code size increase is an acceptable trade-off if the resulting solution has lower WCET and/or energy consumption objective values. For all benchmarks, we observed that the WCET and energy objectives of the solutions on the final PF_T obtained by MO_C were better than the BL values.

From overall evaluations, we calculated the total number of solutions provided by all optimization runs on the final PF_T and averaged it over all the benchmarks. Table 10.1 shows the percentage of the number of solutions from the respective approach's PF_F that ended on the final PF_T, on average, for each optimization. The MO_D–FPA and MO_D–SPEA optimization runs provided 5.29% and 10% solutions on the final PF_T, respectively. The BL evaluation run provided 50% solutions on the final PF_T from its PF_F, i.e., for 10 benchmarks out of 20, the BL solution ended up on the final PF_T. As mentioned in Sections 5.4.2, 6.5.3, 8.5.2, and 9.5.3.1, Table 10.1 can skew numbers towards single-solution-producing approaches. Consequently, even though MO_C–FPA provided a higher number of solutions from its PF_F to the final PF_T, Table 10.1 showed only a 32.97% contribution from MO_C–FPA. Overall, MO_C provided most solutions on the Pareto front PF_T, and the SPEA algorithm contributed more solutions than FPA on the final PF_T for MO_C.

We also compared the reduction in WCET and energy consumption objectives achieved utilizing MO_D and MO_C over the baseline BL evaluation. For brevity's sake, we present the code size-related evaluation details in Subsection 10.5.3. Table 10.2 reports the percentage reduction in the objective values obtained using the optimization approaches over BL. In Table 10.2, percentage reductions are computed as the reduction of average objective values across all benchmarks, relative to the BL evaluations. From Table 10.2, we observe that the reduction in WCET is higher than that in energy consumption for both MO_D and MO_C. MO_D–FPA and MO_D–SPEA both provided a higher average reduction in WCET and energy consumption compared to MO_C–FPA. However, MO_C–SPEA performed better than both MO_D–FPA and MO_D–SPEA in terms of average WCET and energy consumption reductions.

To dynamically copy address objects from Flash to ISPM, MO_D uses the *memcpy()* function, and to dynamically decompress objects from the data section to ISPM, MO_C uses the `decomp()` function. The time taken by *memcpy()* to dynamically allocate address objects depends on their sizes. Similarly, the time taken by the `decomp()` function to decompress objects depends on their sizes as well. The FastLZ decompression used in this chapter is asymmetric, i.e., the decompression is lightweight compared to the compile-time compression. However, unlike the DMA-aware approach from Chapter 9, we cannot guarantee that the `decomp()` function will always take less time than the *memcpy()* function. The *memcpy()* function directly copies the address objects from Flash to ISPM, whereas the `decomp()` function needs to decompress the compressed objects from the data section to ISPM. Therefore, we can argue that the *memcpy()* function might be faster than `decomp()` for the same-sized objects. However, unlike *memcpy()*, the `decomp()` function deals with compressed objects. Moreover, the reduction in WCET and energy objectives achieved using MO_C over MO_D could be attributed to the random nature of the metaheuristic algorithms.

It is important to note that, unlike MO_C, MO_D can cause an uncontrolled increase in code size, as evident from Chapter 8. Furthermore, from the Pareto optimal solutions (cf. Figures 10.7, G.1, and G.2) and Table 10.1, we can see that MO_D provided the least number of solutions on the final PF_T. It implies that the code size reduction achieved due to MO_C improved the MO_C's solutions overall quality compared to the MO_D's solutions. In Subsection 10.5.3, we further compare the code sizes of the solutions obtained using MO_D, MO_C, and BL.

10.5.3 Code Size Comparison

In this subsection, we compared the code size increase or decrease achieved due to compiler-level code transformations needed during optimizations. In MO_C, we do not constrain the growth in code size, so an increase in the code

Figure 10.8.: Increase in code size comparison (in %) between MO_D code size and compressed and uncompressed versions of the MO_C code relative to the BL code (0% is the BL code size)

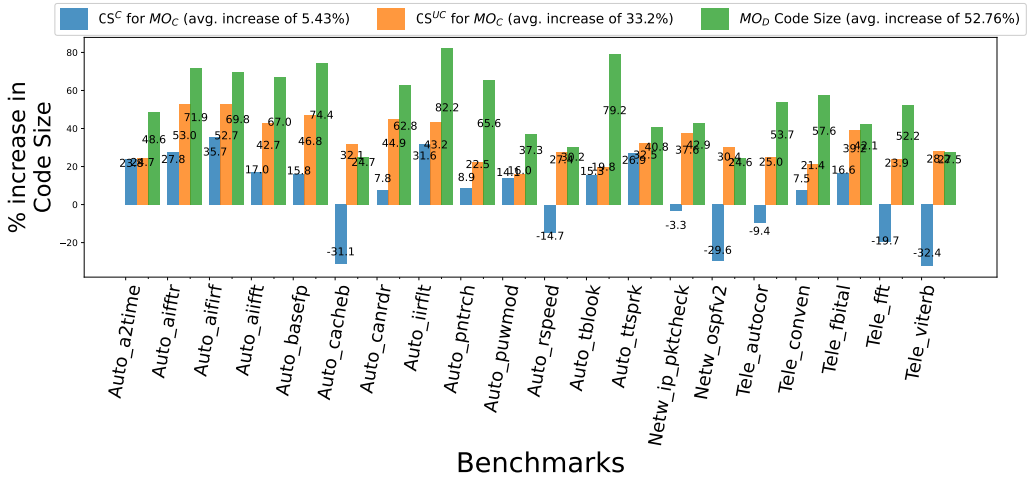


Table 10.3.: Average increase in code size due to MO_D and MO_C over BL

MO_D		MO_C	
FPA	SPEA	FPA	SPEA
51.53	53.99	6.99	3.87

size compared to the BL code size is acceptable if the resulting solution has lower WCET, energy objectives, or both.

Table 10.3 presents the average percentage increase in code size due to MO_D and MO_C over the BL code size. From the table, we can see that both MO_D and MO_C resulted in, on average, an increase in code size. However, both MO_C–FPA and MO_C–SPEA resulted in a smaller increase in code size over BL compared to MO_D–FPA and MO_D–SPEA. Moreover, MO_C–SPEA resulted in the least average increase in code size. Therefore, from Tables 10.2 and 10.3, we can say that MO_C–SPEA performed better than MO_C–FPA, MO_D–FPA, and MO_D–SPEA.

Figure 10.8 shows the percent increase in code size comparison between the MO_D code size, the compressed MO_C code size CS^C (cf. Definition 10.2), and the uncompressed MO_C code size CS^{UC} (cf. Definition 10.3) relative to the BL code size. In this figure, we average the CS^C and CS^{UC} results for MO_C–FPA and MO_C–SPEA together for each benchmark. Similarly, we average the MO_D–FPA and MO_D–SPEA together for each benchmark to represent them in the figure.

The MO_D code size only consists of the code in the Flash memory, and the code sizes CS^C and CS^{UC} are related to the MO_C optimization (cf. Section 10.3.4). Although CS^{UC} has no bearing on the MO_C optimization, we calculate CS^{UC} to evaluate the decrease in code size achieved due to the compression of all the

compression objects $\mathbb{CO}_{(p,m,c)}$. On average, $\texttt{CS}^{\texttt{C}}$ constitutes a code size 25.48% smaller than that of $\texttt{CS}^{\texttt{UC}}$.

From Figure 10.8, we can infer that, for all benchmarks, $\texttt{CS}^{\texttt{UC}}$ and $\mathrm{MO_D}$ code sizes are higher than $\texttt{CS}^{\texttt{C}}$. $\texttt{CS}^{\texttt{UC}}$ and $\mathrm{MO_D}$ resulted in a 33.2% and 52.76% increase in code size compared to the BL code size. However, we got only a 5.43% increase in the $\texttt{CS}^{\texttt{C}}$ code size, on average, than the BL code size due to the compression awareness of the proposed $\mathrm{MO_C}$ optimization.

To perform dynamic decompression, we need to perform code transformations to correct jumps between basic blocks, add a decompression function (`decomp()`) to the code, and add `decomp()` function calls that dynamically decompress compression objects in SPM during runtime. These code transformations could cost a lot in terms of code size. For example, for the `Auto_aiifft` benchmark, the original (BL) code size is 2590 bytes, and the $\texttt{CS}^{\texttt{UC}}$ code size of a solution on its $\mathrm{PF_F}$ is 3372 bytes. The code size of the FastLZ decompression call is 340 bytes for our ARM Cortex-M0 compiled code. Therefore, the increase in code size due to code transformations is $(3372 - 2590 - 340) = 442$ bytes. The compressed $\texttt{CS}^{\texttt{C}}$ code size of the same solution is 2710 bytes. The average 5.43% increase in the $\texttt{CS}^{\texttt{C}}$ code size compared to the BL code size indicates that the extra code added to generate compression-aware dynamically allocated code is still, on average, more than the reduction in code size achieved due to compression.

However, by using the $\mathrm{MO_C}$ optimization approach, we can mitigate the code size and perform dynamic decompression simultaneously, leading to a decrease in WCET and energy consumption as well. For 7 benchmarks, we got final solutions on $\mathrm{PF_T}$ with code sizes smaller than the BL code (cf. Figure 10.8). For the rest, we saw an acceptable growth in the code size. However, if we use $\mathrm{MO_D}$ instead of $\mathrm{MO_C}$, we observe a higher increase in code size. Even though there was an average increase in code size using $\mathrm{MO_C}$–FPA and $\mathrm{MO_C}$–SPEA, we achieved a significant average reduction in WCET and energy compared to the BL evaluations (cf. Tables 10.2 and 10.3).

$\mathrm{MO_C}$ does not guarantee that its solutions will always have better WCET and energy consumption objectives than $\mathrm{MO_D}$ (cf. Table 10.2). However, $\mathrm{MO_C}$ provided, on average, a 50.49% reduction in code size for all benchmarks compared to $\mathrm{MO_D}$. All the solutions provided by $\mathrm{MO_C}$ for all benchmarks had better $\texttt{CS}^{\texttt{C}}$ values than the $\mathrm{MO_D}$'s code size. Therefore, although we cannot guarantee that we will always achieve better WCET and energy consumption using $\mathrm{MO_C}$ over $\mathrm{MO_D}$, we can certainly say that $\mathrm{MO_C}$ will always provide reductions in code sizes compared to the $\mathrm{MO_D}$ solutions.

10.5.4 Quality Indicators

To further analyze the quality of the obtained solutions in this subsection, we consider four quality indicators—*Coverage* ($\mathcal{C}$) (cf. Definition 4.6), *Non-dominance Ratio* (NR) (cf. Definition 4.7), *Ratio of Non-dominated Solutions* (RNS) (cf.

Table 10.4.: Total number of benchmarks with best (or nondominated) quality indicators for each optimization approach

Quality Metrics	MO_D		BL	MO_C	
	FPA	SPEA		FPA	SPEA
$\mathcal{C}$	1 (1)	0 (1)	7 (3)	1 (1)	7 (2)
NR	0 (0)	0 (1)	0 (0)	3 (2)	14 (3)
RNS	1 (1)	0 (1)	7 (3)	1 (1)	7 (2)
HV	0 (0)	0 (0)	1 (0)	7 (3)	9 (3)

Definition 4.8), and *HyperVolume* (HV) (cf. Definition 4.9). As the exact Pareto front is unknown, we consider the final Pareto front PF_T the best available approximation of the exact Pareto front to calculate the values for the quality indicators. For reference sake, Tables G.1 and G.2 in Appendix G provide the quality indicator values for all the benchmarks from these evaluations.

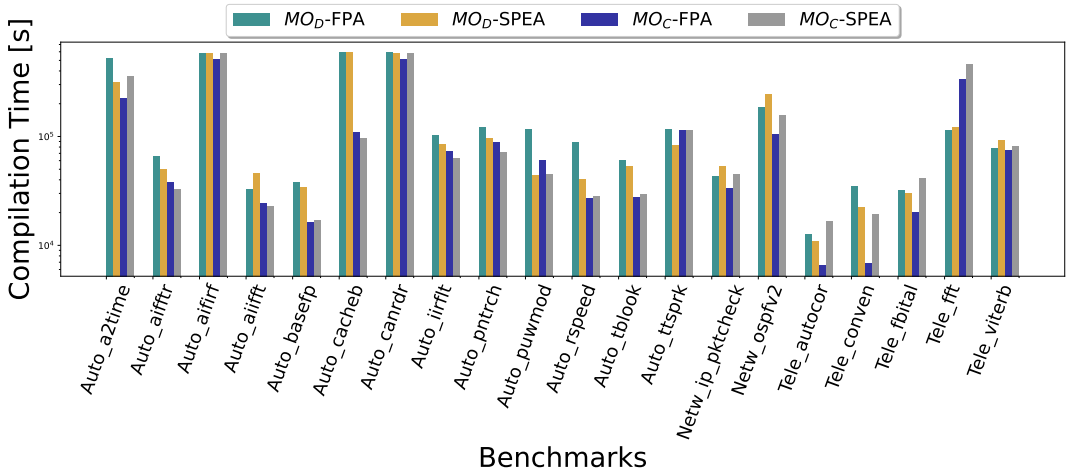
Table 10.4 provides the total number of benchmarks with best or nondominated $\mathcal{C}$, NR, RNS, and HV quality indicators for each optimization approach. Similar to quality indicator evaluations in previous chapters, for Table 10.4, the numbers outside the brackets represent the total number of benchmarks with the best quality indicators for the respective approach compared to others. The numbers inside the brackets shown in the table represent the number of benchmarks with nondominated solutions compared to other techniques. For example, let MO_C and MO_D have 5 solutions on respective PF_F and provide 1 solution each on the final PF_T. Then, both approaches have the same quality indicator values, and benchmarks with such quality indicators are quantified in the table by the value inside the bracket.

From Table 10.4, we can see that MO_C provided better quality solutions compared to MO_D for most benchmarks. However, the BL evaluation performed slightly better than MO_C–SPEA in terms of $\mathcal{C}$ and RNS. As mentioned in the previous subsection, BL has one solution by definition. If this BL's solution is on the final PF_T, then $\mathcal{C}$ is equal to zero, and RNS is one (the best possible quality indicator values). However, from Figures 10.7, G.1, and G.2, we can see no benchmarks where both MO_C–FPA and MO_C–SPEA have failed to provide solutions to the final PF_T along with BL.

Moreover, Table 10.4 clearly shows that MO_C–FPA and MO_C–SPEA performed better than BL in terms of NR and HV metrics. From the NR metric results, we can see that MO_C provided a higher number of solutions on the final PF_T than the other approaches. From the HV metric results, we can see that the MO_C solutions encompassed the highest volume in the objective space compared to the other approaches.

For one benchmark, MO_D–FPA performed better than MO_C and BL in terms of $\mathcal{C}$ and RNS metrics. I.e., for the Tele_fbital benchmark, even though MO_C provided more solutions on the final PF_T, MO_D–FPA had 1 solution on PF_F,

Figure 10.9.: Compilation times for MO_D and MO_C optimization runs



which was on the final PF_T. It resulted in the best $\mathcal{C} = 0$ and `RNS` $= 1$ metric values for `Tele_fbital` (cf. Figure G.2). For `Auto_puwmod`, the BL solution had a slightly higher HV metric than MO_D (cf. Table G.2).

However, for the rest of the benchmarks, MO_C provided better or nondominated quality solutions compared to MO_D and BL. Moreover, for MO_C, SPEA outperformed FPA in terms of all quality indicators.

10.5.5 Compilation Times

In this subsection, we compare the compilation time required to solve the MO_C and MO_D optimizations with each other. As we need to perform only one WCET, energy, and code size evaluation for BL, we skip the BL's compilation time comparison with MO_C and MO_D for brevity's sake. Figure 10.9 presents the compilation times required by MO_C and MO_D in the form of a bar plot. The x-axis represents all the evaluated benchmarks from `EEMBC`, and the y-axis shows the compilation time required to finish the optimizations.

MO_C and MO_D are both time-consuming optimizations. As mentioned in Subsection 8.5.4, MO_D–FPA and MO_D–SPEA took 48.89 h and 44.07 h, on average, to finish their respective runs. MO_C–FPA and MO_C–SPEA, on average, took 33.46 h and 39.62 h to finish the optimization, respectively.

In Subsection 8.5.1, we calculated the maximum number of WCET and energy consumption evaluations to complete the MO_D optimization run. Similarly, during the evaluations in this chapter, we perform a maximum of 10000 objective evaluations for the MO_C optimization run. Moreover, we perform code size evaluations for each individual; however, the code size evaluations are not time-consuming compared to WCET and energy evaluations. Compared to MO_D, the code generation process for MO_C optimization requires a few extra steps for each individual: compressing loops using the FastLZ library, running Algorithm 10.2, and compressing functions using the FastLZ library.

However, MO_C was quicker than MO_D and, on average, took 21.39% less compilation time during these evaluations.

The time required for WCET and energy consumption evaluations is the major compilation time bottleneck for the MO_D and MO_C optimization runs. As the compilation times needed by MO_C and MO_D largely depend on the number of evaluations required by the optimization to come to fruition, it is difficult to conclude that MO_C will always be faster than MO_D due to the non-deterministic nature of the metaheuristic algorithms. Moreover, from Figure 10.9, we can clearly see that MO_D required less compilation time than MO_C for some benchmarks.

10.6 Conclusion

In this chapter, we proposed a compiler-level compression-aware multi-objective dynamic SPM allocation optimization problem (MO_C) and solved it using two metaheuristic algorithms—SPEA and FPA. The Pareto optimal compression-aware dynamically allocatable code is generated by compressing compression objects at compile-time and storing them in the data section. During runtime, the compressed objects are decompressed into ISPM by a `decomp()` function call and executed from ISPM. Evaluations compared the proposed MO_C with MO_D and BL.

Due to strict constraints on compression ratio (CR) and static SPM size, SO_F was unable to optimize code for any EEMBC benchmarks. While MO_D successfully minimized WCET and energy consumption, it incurred a 52.76% increase in code size compared to the BL run. However, MO_C provided the highest quality solutions on the final Pareto front with a controlled increase of 5.43% in code size. Due to MO_C's compression awareness, the MO_C solution had an average 25.48% reduction in its CS^C code size compared to CS^{UC}. Furthermore, MO_C–SPEA achieved the most—37.46% and 13.21%—reduction in terms of WCET and energy consumption compared to the BL, respectively.

Multiple WCET and energy analyses are required to solve the MO_C optimization problem. Moreover, we need to solve the address assignment problem and perform compression using the FastLZ library to generate compression-aware code for every individual in all populations. As observed during the evaluations conducted in this chapter, it leads to a time-consuming MO_C optimization process.

Part IV

Conclusion

11

Conclusion and Future Work

11.1 Summary

This thesis proposed multiple compiler-level multi-objective SPM allocation-based optimizations and solved them using two metaheuristic algorithms—FPA and SPEA. We compared the proposed optimizations with state-of-the-art ILP-based single-objective memory allocation-based optimizations. We considered two or three objectives simultaneously—WCET, energy consumption, schedulability, and code size—while solving multi-objective optimizations throughout the thesis. All the optimizations proposed in this thesis and the state-of-the-art optimizations are implemented and evaluated within the WCC compiler framework.

In Chapter 5, we presented a multi-objective static SPM allocation optimization for a single-task real-time system to minimize WCET and energy consumption simultaneously. The proposed multi-objective optimization is solved using both FPA and SPEA algorithms. We compared the proposed optimization with the state-of-the-art single-objective ILP-based static SPM allocation optimization and a baseline evaluation. We showed that multi-objective optimization provides superior quality Pareto optimal solutions compared to the single-objective optimization and the baseline. However, the proposed metaheuristic algorithm-based multi-objective optimization is time-consuming, i.e., iteratively performing WCET and energy analyses creates a huge bottleneck.

In Chapter 6, we proposed a novel approximation model to approximate WCET and energy objectives while performing multi-objective static SPM allocation to resolve the compilation time bottleneck. We showed that the proposed approximation model can approximate WCET and energy objectives with 4%-7% approximation errors and maintain the Pareto solution quality compared to the original multi-objective optimization approach from Chapter 5. Moreover, we observed a 94.89% reduction in compilation time using the approximation model to approximate WCET and energy objectives, on average, over the traditional respective analyzers.

In Chapter 7, we expanded the multi-objective static SPM allocation for multi-task systems (MTMO). We aimed to simultaneously minimize WCET, energy consumption, and schedulability objectives of a program using the proposed

optimization. In this chapter, we propose a Path-based Constraint Approach (PCA) and an Impact-based Constraint Approach (ICA) to constrain the large solution space of the multi-task multi-objective optimization problem. On average, PCA and ICA provided a 23.93% and 24.66% reduction in compilation time over the MTMO approach, respectively. Moreover, we achieved an average reduction of 23.97% and 52.83% in search space dimensions using PCA and ICA, respectively, compared to MTMO. However, PCA provided worse quality solutions than the original and the ICA techniques. On average, ICA–FPA provided the best quality Pareto optimal solutions compared to the other approaches.

In Chapter 8, we proposed a novel multi-objective dynamic SPM allocation optimization to optimize the program's WCET and energy consumption simultaneously. Static SPM allocation has an SPM size constraint, and we circumvent this constraint by introducing dynamic allocation during runtime. Dynamic allocation requires more code transformations compared to static allocation. Along with jump corrections, we need to add memory copy ($memcpy()$) function calls to enable dynamic allocation of memory objects during runtime. We incurred extra WCET, energy, and code size costs due to these code transformations and costly $memcpy()$ function calls.

In Chapter 9, we presented a DMA-aware multi-objective dynamic SPM allocation optimization that uses function calls to DMA instead of the CPU to perform dynamic allocation during runtime. We expanded the WCC framework to handle compilation and analysis for an architecture with a DMA controller. We proposed a compiler-level DMA model and an accompanying DMA analyzer to calculate the total execution time cost for DMA transfers in arbitration mode. We also proposed a single-objective DMA Call Placement Optimization (DCPO) that aims to find optimal places within the program to place DMA calls such that the total execution time required for DMA transfer is minimized. We solved the DCPO problem using ILPs, FPA, and SPEA, and showed that the ILP-based DCPO performed best for all benchmarks. We also showed that the DMA-aware multi-objective optimization performed better than CPU-based multi-objective optimization for all benchmarks. However, due to the difference between the granularity of basic block- and memory object-level allocations, sometimes static allocation-based multi-objective optimization provided better solutions. But, on average, the DMA-aware multi-objective optimization generated better Pareto optimal solutions both qualitatively and quantitatively.

In Chapter 10, we presented a compression-aware multi-objective dynamic SPM allocation (MO_C) that optimizes the program's WCET, energy consumption, and code size simultaneously. We integrated compression techniques with the dynamic allocation technique to curb the code size increase due to the dynamic allocation of memory objects. During evaluations, we showed that the proposed compression-aware technique can decrease the program's code size and increase the overall solution quality compared to the original multi-

objective dynamic SPM allocation technique from Chapter 8. MO_C achieved a 25.48% average reduction in its own code size due to the compression of memory objects. We observed only an average increase of 5.43% in code size due to MO_C compared to the baseline code. However, MO_C achieved an average reduction of 29.53% and 7.51% in WCET and energy consumption compared to the baseline evaluation.

11.2 Future Work

Based on the insights and evaluations from this thesis, various topics can be promising directions for future work:

Metaheuristic Algorithms:

In this thesis, we use two metaheuristic algorithms—FPA and SPEA. The choice of metaheuristic algorithms to solve memory allocation-based optimizations is not the focus of this thesis. From all the evaluations conducted in this thesis, we can see the following: SPEA contributed, on average, better solutions to the final Pareto front PF_T for the MO_S, MO_{Approx}, and MO_C optimization techniques. FPA contributed, on average, better solutions to PF_T for the ICA optimization. For the MTMO, PCA, MO_D, and MO_{DMA} techniques, both FPA and SPEA performed equally well. Therefore, we could say, on average, SPEA performs better for more approaches than FPA during these evaluations.

However, metaheuristic algorithms are non-deterministic, and some metaheuristics can perform better for some problems and worse for others. Therefore, future work can focus on a comprehensive analysis of different metaheuristic algorithms for memory allocation-based optimizations. Moreover, the performance of metaheuristic algorithms depends on the hyperparameter values used during optimization. Future work could be carried out that focuses on a comprehensive evaluation of hyperparameter tuning for memory allocation-based optimizations.

Scalability Problems:

The multi-objective memory allocation-based optimizations proposed in this thesis are time-consuming. The requirement of multiple objective analyses acts as a compilation time bottleneck. In Chapter 6, we proposed an approximation model to deal with this bottleneck for static SPM allocation-based multi-objective optimization successfully. However, the evaluations performed for dynamic allocation-based optimizations show the same compilation time bottleneck. Unlike static SPM allocation from Chapter 6, the dynamic allocation-based optimizations proposed in this thesis dynamically allocate functions and loops. As the approximation model from Chapter 6 relies on the basic block-level static allocation-based decision variables, we cannot directly use this

model for approximating WCET and energy consumption for the dynamically allocated code. Moreover, an approximation model for the dynamically allocated code should approximate the objective costs incurred due to the dynamic copying of memory objects as well. Future work can focus on approximating WCET and energy objectives for multi-objective dynamic SPM allocation.

PCA and ICA reduce the MTMO problem's search space using the program's WCET and energy consumption-related properties. As mentioned in Section 7.7, PCA and ICA do not constrain the global data objects. Future work could expand PCA and ICA to identify unnecessary global data objects and further reduce the optimization problem's search space.

Contrary to dynamic allocation-based optimizations, the search space of the multi-objective static SPM allocation is defined by basic blocks. Therefore, PCA and ICA cannot be used directly for dynamic allocation-based optimization. Future work could be carried out on the scalability issue of the dynamic allocation-based multi-objective optimization.

If the scalability problem is resolved, future work could perform large evaluation-based studies that compare the performance of various metaheuristic algorithms for compiler-level memory allocation-based optimizations. We can use these approximation models to approximate WCET and energy objectives faster than performing time-consuming analyses, and a PCA and ICA-like approach to reduce the problem's search space. This kind of study could help us find the best-performing metaheuristics to solve multi-objective memory allocation problems and tune their respective hyperparameters as well.

Multi-Task and Multi-Core Systems:

We considered a multi-task system and schedulability as an objective while performing a multi-objective static SPM allocation technique. Future work can increase the complexity of the static SPM allocation-based problem by considering multi-core systems with multiple tasks running on each core.

In this thesis, we deal with single-task single-core systems while performing dynamic allocation-based optimizations. However, similar to MTMO, the MO_D optimization problem could be easily expanded for a non-preemptive multi-task system. As no task is preempted by another, in this case, the MO_D problem does not need to go through any major change. Moreover, the implementation of the MO_D optimization within WCC already supports such a non-preemptive multi-task setup.

However, for preemptive multi-task systems, we need to change the dynamic allocation strategy. For example, let us assume that task `A` is preempted by task `B`. In this case, task `B` can dynamically allocate objects to SPM and overwrite the objects from task `A`. Therefore, future work needs to consider how to store the dynamically allocated state of preempted task `A` before the execution of task `B`.

Moreover, future work can consider a multi-task multi-core system in the context of a multi-objective dynamic SPM allocation optimization problem. Similar to the multi-task case, the current version of MO_D could be easily extended for multi-core systems with separate SPMs for each core. Future work could focus on the dynamic allocation of memory objects for multi-core systems with shared memories.

Memories:

The static and dynamic allocation problems proposed in this thesis are proposed for an architecture with the Flash memory and the SPM. However, these optimizations can easily be expanded for architectures that support more complicated memory hierarchies. For the Flash and SPM memory setup considered in this thesis, we use the binary decision variables to decide in which memory a particular basic block or memory object will reside. For more complicated memory hierarchies, integer-valued decision variables could be helpful. However, the metaheuristic algorithms used in this thesis are designed for binary decision variables. Therefore, for integer-valued decision variables, we need to consider appropriate metaheuristics as well. Future work can consider a more complicated memory hierarchy and explore the memory allocation-based multi-objective optimizations proposed in this thesis.

Dynamic Allocation and Liveness Analysis:

The dynamic allocation and accompanying liveness analysis proposed in this thesis have a few limitations. In this thesis, we do not place $memcpy()$ function calls inside loops. Consequently, as explained using the example in Figure 8.5 in Section 8.3.2, we cannot place two non-conflicting loops within the same loop nest at overlapping memory addresses in the SPM. The current version of the liveness analysis conservatively considers all loops within a loop nest to be in conflict with each other. In future work, the liveness analysis could be extended to detect such non-conflicting loops within a loop nest. This extension could enable the placement of $memcpy()$ calls inside a loop nest when necessary.

However, placing $memcpy()$ calls inside a loop increases WCET and energy consumption, as these function calls are triggered during every iteration of the loop. Since $memcpy()$ calls are executed by the CPU, there is no benefit in applying a DCPO-like optimization to determine their placement. In particular, placing $memcpy()$ calls further away from the *def* basic block of the memory object does not provide any advantage. Therefore, a future extension could introduce a heuristic or analytical approach to determine whether placing $memcpy()$ calls inside a loop nest is beneficial.

Moreover, in Chapter 9, DMA calls are also not placed inside loops. In future work, the DCPO optimization could be further extended and combined with the enhanced liveness analysis to accommodate the placement of DMA

calls inside loops by relaxing the loop constraint defined by Equation 9.13 in Subsection 9.3.3.

DMA-Aware and Compression-Aware Topics:

In this thesis, we considered single-ported memories for the proposed DMA-aware dynamic allocation approach. Due to the use of single-ported memories, the CPU and the DMA controller can operate with limited parallelism. Future work could consider underlying architectures with dual-ported memories, thereby removing contention between the CPU and the DMA controller.

Moreover, in Appendix F, we proposed a theoretical pipeline-aware DMA cost model for an underlying 3-stage pipeline architecture. However, the practical implementation of this model within the WCC compiler, along with an extensive comparison against the pipeline-unaware model from Section 9.3.2, is still missing. Future work could extend the DMA cost calculation model within WCC to incorporate this pipeline-aware model and compare it with the existing pipeline-unaware model. This extension would allow a system designer to choose between different cost models when solving the DCPO optimization problem within WCC.

Furthermore, in this thesis, we used the FastLZ library to perform compile-time compression and runtime decompression. Further investigation could explore alternative compression strategies to achieve higher degrees of compression and further reduce the code size. In addition, we considered only compression and decompression of code memory objects in this thesis. Future work can explore data compression techniques alongside code compression for real-time systems.

This thesis considered a special hardware component (a DMA controller) capable of dynamically allocating memory objects faster than the CPU during runtime. Future work can explore hardware-supported runtime decompression mechanisms that can dynamically decompress memory objects more efficiently than CPU-based software approaches. Similar to the DMA controller-based approach, such work could develop a compile-time hardware compressor model to estimate the corresponding runtime execution time costs.

Pareto Optimal Solutions:

In this thesis, we observed that the proposed multi-objective optimizations provided a large quantity of Pareto optimal solutions with visible trade-offs between multiple objectives. However, in this thesis, we do not focus on the choice of the solutions present on the Pareto front. The system designer is presented with a choice of multiple solutions after the metaheuristic algorithms have finished performing the optimization.

Future work can add application-specific constraints to the multi-objective optimization itself. It could further constrain the solution space and might reduce the time required for the optimization to come to fruition. However,

this approach might also provide multiple solutions on the final Pareto front that fulfill the provided constraint. Future work can also focus on the design of a so-called decision-maker based on specified criteria that help the system designer weed out, sort, classify, or choose the best trade-offs between the solutions on the final Pareto front.

Multi-Objective Optimizations:

This thesis focuses on memory allocation-based multi-objective optimizations. We proposed compiler-level multi-objective static and dynamic allocation techniques. However, the problem formulations for the proposed optimizations in this thesis have some generic and some problem-specific parts towards the underlying memory allocation techniques. For example, the mathematical formulations for the minimization problems look similar. However, the meaning of decision variables changes, and the optimization constraints also look different depending on the underlying memory allocation problem.

Future work can consider other compiler-level optimizations, like register allocation, cache locking, etc., and extend them as multi-objective optimizations utilizing the multi-objective optimization framework from this thesis. Moreover, future work could focus on unifying and generalizing such compiler-level optimizations under one multi-objective optimization framework.

Bibliography

[AB15] Kapil Anand and Rajeev Barua. Instruction-cache locking for improving embedded systems performance. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(3):1–25, 2015.

[Abs] AbsInt Angewandte Informatik GmbH. XTC: XML Timing Cookie Language. `http://www.absint.com/xtc/`.

[Abs24] AbsInt Angewandte Informatik, GmbH. aiT Worst-Case Execution Time Analyzers, 2024.

[AGLS16] Peter Altenbernd, Jan Gustafsson, Björn Lisper, and Friedhelm Stappert. Early execution time-estimation through automatically generated timing models. *Real-Time Systems*, 52:731–760, 2016.

[AMF+04] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratch-pad mapping of code. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 259–267, 2004.

[App04] Andrew W Appel. *Modern compiler implementation in C*. Cambridge university press, 2004.

[ARM09a] ARM Ltd. Cortex-m0 devices generic user guide, 2009. ARM DUI 0497.

[ARM09b] ARM Ltd. Cortex-m0 technical reference manual, 2009. ARM DDI 0432C.

[BAG+17] Florin Balasa, Noha Abuaesh, Cristian V. Gingu, et al. Energy-aware memory management for embedded multidimensional signal processing applications. *EURASIP Journal on Embedded Systems*, 6(1), 2017.

[Bar03] Sanjoy K. Baruah. Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks. *Real-Time Systems*, 24(1):93–128, 2003.

[BB05] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-time systems*, 30(1):129–154, 2005.

[BC15] Anuradha Balasundaram and Vivekanandan Chenniappan. Optimal code layout for reducing energy consumption in embedded systems. In *2015 International Conference on Soft-Computing and Networks Security (ICSNS)*, pages 1–5. IEEE, 2015.

[BCDMS17] Armelle Bonenfant, Denis Claraz, Marianne De Michiel, and Pascal Sotin. Early wcet prediction using machine learning. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[BDC+07] Erik Brockmeyer, Bart Durinck, Henk Corporaal, et al. Layer assignment techniques for low energy in multi-layered memory organizations. In *Designing Embedded Processors*, pages 157–190. Springer, 2007.

[BTW13] Prasanna Balaprakash, Ananta Tiwari, and Stefan M Wild. Multi objective optimization of hpc kernels for performance, power, and energy. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 239–260. Springer, 2013.

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[CPI+05] Antonio Marti Campoy, Isabelle Puaut, Angel Perles Ivars, et al. Cache contents selection for statically-locked instruction caches: An algorithm comparison. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 49–56, 2005.

[DBD+06] Minas Dasygenis, Erik Brockmeyer, Bart Durinck, Francky Catthoor, Dimitrios Soudris, and Adonios Thanailakis. A combined dma and application-specific prefetching approach for tackling the memory latency bottleneck. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(3):279–291, 2006.

[DLM14] Huping Ding, Yun Liang, and Tulika Mitra. WCET-centric dynamic instruction cache locking. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2014.

[Dor05] Informatik Centrum Dortmund. *ICD Low-Level Intermediate Representation backend infrastructure (LLIR). Developer Manual*, 2005.

[Dor09] Informatik Centrum Dortmund. *ICD-C Compiler framework. Developer Manual*, 2009.

[DP07] Jean-Francois Deverge and Isabelle Puaut. WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, pages 179–190, 2007.

[ED18] Michael TM Emmerich and André H Deutz. A Tutorial on Multiobjective Optimization: Fundamentals and Evolutionary Methods. *Natural Computing*, 17:585–609, 2018.

[Ehr05] Matthias Ehrgott. *Multicriteria optimization*, volume 491. Springer Science & Business Media, 2005.

[Erm03] Andreas Ermedahl. *A modular tool architecture for worst-case execution time analysis*. PhD thesis, Acta Universitatis Upsaliensis, 2003.

[ES15] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer, 2015.

[FAH+16] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. Taclebench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2016.

[Fal09] Heiko Falk. WCET-aware Register Allocation based on Graph Coloring. In *Proceedings of the 46th Annual Design Automation Conference*, pages 726–731, 2009.

[FK09] Heiko Falk and Jan C Kleinsorge. Optimal static wcet-aware scratchpad allocation of program code. In *Proceedings of the 46th Annual Design Automation Conference*, pages 732–737, 2009.

[FK11] Heiko Falk and Helena Kotthaus. Wcet-driven cache-aware code positioning. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 145–154, 2011.

[FL10] Heiko Falk and Paul Lokuciejewski. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems*, 46(2):251–298, 2010.

[FMA+04] Poletti Francesco, Paul Marchal, David Atienza, et al. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41st Annual Design Automation Conference*, pages 238–243, 2004.

[FPT07] Heiko Falk, Sascha Plazar, and Henrik Theiling. Compile-Time Decided Instruction Cache Locking Using Worst-Case Execution Paths. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 143–148, 2007. DOI 10.1145/1289816.1289853.

[FSS11] Heiko Falk, Norman Schmitz, and Florian Schmoll. Wcet-aware register allocation based on integer-linear programming. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 13–22. IEEE, 2011.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[GCZY21] Chen Guo, Song Ci, Yanglin Zhou, and Yang Yang. A survey of energy consumption measurement in embedded systems. *IEEE Access*, 9:60516–60530, 2021.

[GKCE17] Kyriakos Georgiou, Steve Kerrison, Zbigniew Chamski, and Kerstin Eder. Energy transparency for deeply embedded programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(1):1–26, 2017.

[GT08] Chi-Keong Goh and Kay Chen Tan. A Competitive-Cooperative Coevolutionary Paradigm for Dynamic Multiobjective Optimization. *IEEE Transactions on Evolutionary Computation*, 13(1):103–127, 2008.

[Gur24] Gurobi Optimization, Inc. Gurobi Optimizer, 2024. URL: `https://www.gurobi.com`.

[HE08] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 165–174. ACM, 2008.

[Hid07] Ariya Hidayat. Fastlz-lightning-fast lossless compression library, 2007.

[HMH18] Thomas Huybrechts, Siegfried Mercelis, and Peter Hellinckx. A new hybrid approach on wcet analysis for real-time systems using machine learning. In *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[IIY07] Yuriko Ishitobi, Tohru Ishihara, and Hiroto Yasuura. Code placement for reducing the energy consumption of embedded processors with scratchpad and cache memories. In *2007 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*, pages 13–18. IEEE, 2007.

[JF19] Shashank Jadhav and Heiko Falk. Multi-objective optimization for the compiler of real-time systems based on flower pollination algorithm. In *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems*, pages 45–48, 2019. doi:10.1145/3323439.3323977.

[JF22] Shashank Jadhav and Heiko Falk. Approximating wcet and energy consumption for fast multi-objective memory allocation. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, pages 162–172, 2022. doi:10.1145/3534879.3534889.

[JF23a] Shashank Jadhav and Heiko Falk. Efficient and Effective Multi-Objective Optimization for Real-Time Multi-Task Systems. In *21th International Workshop on Worst-Case Execution Time Analysis (WCET 2023)*, volume 114 of *Open Access Series in Informatics (OASIcs)*, pages 5:1–5:12, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.WCET.2023.5.

[JF23b] Shashank Jadhav and Heiko Falk. Towards Multi-Objective Dynamic SPM Allocation. In *21th International Workshop on Worst-Case Execution Time Analysis (WCET 2023)*, volume 114 of *Open Access Series in Informatics (OASIcs)*, pages 6:1–6:12, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.WCET.2023.6.

[JF25] Shashank Jadhav and Heiko Falk. Compiler-level dma-aware multi-objective dynamic spm allocation. *The International Journal of Time-Critical Computing Systems (Real-Time Systems)*, pages 1–65, 2025. doi:10.1007/s11241-025-09436-w.

[JML06] Ramkumar Jayaseelan, Tulika Mitra, and Xianfeng Li. Estimating the worst-case energy consumption of embedded software. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 81–90. IEEE, 2006.

[KBS17] Yooseong Kim, David Broman, and Aviral Shrivastava. WCET-Aware Function-Level Dynamic Code Management on Scratchpad Memory. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(4):1–26, 2017.

[KC02] Mahmut Kandemir and Alok Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *Proceedings of the 39th Annual Design Automation Conference*, pages 628–633, 2002.

[KCNL08] Vasilios Konstantakos, Alexander Chatzigeorgiou, Spiridon Nikolaidis, and Theodore Laopoulos. Energy consumption estimation in embedded systems. *IEEE Transactions on instrumentation and measurement*, 57(4):797–804, 2008.

[Kel15] Timon Kelter. WCET Analysis and Optimization for Multi-Core Real-Time Systems, 2015.

[KFP+16] Tobias Kuhn, Carlos M Fonseca, Luís Paquete, Stefan Ruzika, Miguel M Duarte, and José Rui Figueira. Hypervolume subset selection in two dimensions: Formulations and algorithms. *Evolutionary Computation*, 24(3):411–425, 2016.

[Kir03] Raimund Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Techni-sche Universität Wien, May 2003.

[Kop11] Hermann Kopetz. *Real-Time Systems*. Springer-Verlag GmbH, 2011.

[KRI+01] Mahmut Kandemir, J Ramanujam, Mary Jane Irwin, et al. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th annual Design Automation Conference*, pages 690–695, 2001.

[KTZ06] Joshua D Knowles, Lothar Thiele, and Eckart Zitzler. A tutorial on the performance assessment of stochastic multiobjective optimizers. *Tik report*, 214, 2006.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*. IEEE, Mar 2004. doi:10.1109/cgo.2004.1281665.

[LBCM97] Charles Lefurgy, Peter Bird, I-C Chen, and Trevor Mudge. Improving code density using compression techniques. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 194–203. IEEE, 1997.

[Leh90] John Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings 11th Real-Time Systems Symposium*, pages 201–209, 1990.

[LF15] Arno Luppold and Heiko Falk. Schedulability aware wcet-optimization of periodic preemptive hard real-time multitasking systems. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, pages 101–104, 2015.

[LF17] Arno Luppold and Heiko Falk. Schedulability-Aware SPM Allocation for Preemptive Hard Real-Time Systems with Arbitrary Activation Patterns. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 1074–1079, Lausanne / Switzerland, March 2017.

[LL73] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[LOF20] Arno Luppold, Dominic Oehlert, and Heiko Falk. Compiling for the worst case: Memory allocation for multi-task and multi-core hard real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(2):1–26, 2020.

[LPF+11] Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. Approximating Pareto optimal compiler optimization sequences—a trade-off between WCET, ACET and code size. *Software: Practice and Experience*, 41(21):1437–1458, 2011.

[LPM00] Charles Lefurgy, Eva Piccininni, and Trevor Mudge. Reducing code size with run-time decompression. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*, pages 218–228. IEEE, 2000.

[Lup20] Arno Luppold. *Schedulability-oriented code optimization of hard real-time multitasking systems*. PhD thesis, Technische Universität Hamburg, 2020.

[LZ15] Yu Liu and Wei Zhang. Scratchpad memory architectures and allocation algorithms for hard real-time multicore processors. *Journal of Computing Science and Engineering*, 9(2):51–72, 2015.

[Man67] Glenn K Manacher. Production and stabilization of real-time task schedules. *Journal of the ACM (JACM)*, 14(3):439–465, 1967.

[Man94] Rosario Nunzio Mantegna. Fast, accurate algorithm for numerical simulation of levy stable stochastic processes. *Physical Review E*, 49(5):4677, 1994.

[Mar11] Peter Marwedel. *Embedded system design: embedded systems foundations of cyber-physical systems*. Springer, 2011.

[MDC14] Renato Mancuso, Roman Dudko, and Marco Caccamo. Light-prem: Automated software refactoring for predictable execution on cots embedded systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10. IEEE, 2014.

[MF21a] Kateryna Muts and Heiko Falk. Predicting objectives on a reduced search space of multiobjective function inlining. In *Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems*, pages 23–28, 2021.

[MF21b] Kateryna Muts and Heiko Falk. Predicting worst-case execution times during multi-criterial function inlining. In *International Conference on Machine Learning, Optimization, and Data Science*, pages 281–295. Springer, 2021.

[Mie12] Kaisa Miettinen. *Nonlinear multiobjective optimization*, volume 12. Springer Science & Business Media, 2012.

[MLF18] Kateryna Muts, Arno Luppold, and Heiko Falk. Multi-criteria compiler-based optimization of hard real-time systems. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, pages 54–57, 2018.

[MLF19] Kateryna Muts, Arno Luppold, and Heiko Falk. Compiler-based Code Compression for Hard Real-Time Systems. In *Software and Compilers for Embedded Systems (SCOPES)*, pages 72–81, 2019.

[Muc98] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1 edition, 1998.

[MZLH15] Nikita Mishra, Huazhe Zhang, John D Lafferty, and Henry Hoffmann. A probabilistic graphical model-based approach for minimizing energy under performance constraints. *ACM SIGARCH Computer Architecture News*, 43(1):267–281, 2015.

[NGC+22] Kris Nikov, Kyriakos Georgiou, Zbigniew Chamski, Kerstin Eder, and Jose Nunez-Yanez. Accurate energy modelling on the cortex-m0 processor for profiling and static analysis. In *2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 1–4. IEEE, 2022.

[NYP22] Vincent Nélis, Patrick Meumeu Yomsi, and Luís Miguel Pinho. Timing analysis methodology. In *High Performance Embedded Computing*, pages 113–144. River Publishers, 2022.

[OCKK06] Ozcan Ozturk, Guangyu Chen, M Kandemir, and Ibrahim Kolcu. Compiler-guided data compression for reducing memory consumption of embedded applications. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 814–819, 2006.

[OHRC09] Haluk Ozaktas, Karine Heydemann, Christine Rochange, and Hugues Cassé. Impact of code compression on estimated worst-case execution times. In *17th International Conference on Real-Time and Network Systems*, pages 55–66, 2009.

[OKD+04] Ozcan Ozturk, Mahmut Kandemir, I Demirkiran, et al. Data compression for improving SPM behavior. In *DAC*, pages 401–406, 2004.

[OLF16] Dominic Oehlert, Arno Luppold, and Heiko Falk. Practical challenges of ilp-based spm allocation optimizations. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, pages 86–89. ACM, 2016.

[PA06] Isabelle Puaut and Alexis Arnaud. Dynamic instruction cache locking in hard real-time systems. In *Proc. of the 14th Int. Conference on Real-Time and Network Systems*, 2006.

[Pav07] Ilya Pavlyukevich. Lévy flights, non-local search and simulated annealing. *Journal of Computational Physics*, 226(2):1830–1844, 2007.

[PBB+11] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, 2011.

[PCLGO09] Jason A Poovey, Thomas M Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the eembc benchmark suite. *IEEE micro*, 29(5):18–29, 2009.

[PD02] Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 114–123, 2002.

[PFV+07] Robert Pyka, Christoph Faßbach, Manish Verma, et al. Operating System Integrated Energy Aware Scratchpad Allocation Strategies for Multiprocess Applications. In *Proceedingsof the 10th International Workshop on Software & Compilers for Embedded Systems*, pages 41–50, 2007.

[PKM+12] Sascha Plazar, Jan C Kleinsorge, Peter Marwedel, et al. WCET-aware Static Locking of Instruction Caches. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 44–52, 2012.

[PW07] Shlomit S Pinter and Israel Waldman. Selective code compression scheme for embedded systems. In *Transactions on High-Performance Embedded Architectures and Compilers I*, pages 298–316. Springer, 2007.

[QCM+17] Meikang Qiu, Zhi Chen, Zhong Ming, et al. Energy-Aware Data Allocation With Hybrid Memory for Mobile Cloud Systems. *IEEE Systems Journal*, 11(2), 2017.

[RBE+23] Benjamin Rouxel, Christopher Brown, Emad Ebeid, Kerstin Eder, Heiko Falk, Clemens Grelck, Jesper Holst, Shashank Jadhav, Yoann Marquer, Marcos Martinez De Alejandro, et al. The teamplay project: Analysing and optimising time, energy, and security for cyber-physical systems. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023. doi:10.23919/DATE56975.2023.10137198.

[RS03] Montserrat Ros and Peter Sutton. Compiler optimization and ordering effects on vliw code compression. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 95–103, 2003.

[RYDSP15] Douglas Rodrigues, Xin-She Yang, André Nunes De Souza, and João Paulo Papa. Binary flower pollination algorithm and its application to feature selection. In *Recent advances in swarm intelligence and evolutionary computation*, pages 85–100. Springer, 2015.

[ŠBDM99] Tajana Šimunić, Luca Benini, and Giovanni De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 867–872, 1999.

[SG21] Akash Sachan and Bibhas Ghoshal. Learning based compilation of embedded applications targeting minimal energy consumption. *Journal of Systems Architecture*, 116:102116, 2021.

[SKWM01] Stefan Steinke, Markus Knauer, Lars Wehmeyer, and Peter Marwedel. An accurate and fine grain instruction-level energy model supporting software optimizations. In *Proceedings of International Workshop on Power And Timing Modeling, Optimization and Simulation, September 2001*. Citeseer, 2001.

[SMRC05] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. Wcet centric data allocation to scratchpad memory. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10–pp. IEEE, 2005.

[SP17] Muhammad Refaat Soliman and Rodolfo Pellizzoni. WCET-driven dynamic data scratchpad management with compiler-directed prefetching. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, 2017.

[SPG91] Abraham Silberschatz, James L Peterson, and Peter B Galvin. *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc., 1991.

[SS08] Y.N. Srikant and Priti Shankar. *The Compiler Design Handbook*. CRC Press, 2008.

[Sta08] Richard Stallman. *Using the GNU compiler collection: for GCC version 4.3.3*. SoHo Books GNU Press, 2008.

[STM22a] STMicroelectronics. RM0091 Reference manual: STM32F0x1/ STM32F0x2/ STM32F0x8 advanced Arm®-based 32-bit MCUs, 2022. RM0091 Rev 10.

[STM22b] STMicroelectronics. Using the STM32F0/F1/F3/Cx/Gx/Lx Series DMA controller, 2022. AN2548 - Rev 8.

[SWLM02] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 409–415. IEEE, 2002.

[T+36] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[Tar73] Robert Tarjan. Testing Flow Graph Reducibility. In *ACM Symposium on Theory of Computing*, pages 96–107, 1973.

[Tea18] TeamPlay Consortium. Deliverable D3.1 - Report on the TeamPlay Basic Compiler Infrastructure - Version 1.0, 2018.

[Tea20] TeamPlay Consortium. Deliverable D4.5 - Report on Energy Usage Analysis and on Prototype - Version 1.0, 2020.

[TLK02] Kay Chen Tan, Tong Heng Lee, and Eik Fun Khor. Evolutionary Algorithms for Multi-Objective Optimization: Performance Assessments and Comparisons. *Artificial Intelligence Review*, 17(4):251–290, 2002.

[TMW94] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994.

[TZG+11] Hideki Takase, Gang Zeng, Lovic Gauthier, Hirotaka Kawashima, Noritoshi Atsumi, Tomohiro Tatematsu, Yoshitake Kobayashi, Shunitsu Kohara, Takenori Koshiro, Tohru Ishihara, et al. An integrated optimization framework for reducing the energy consumption of embedded real-time applications. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 271–276. IEEE, 2011.

[UB03] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 276–286, 2003.

[VM07] Manish Verma and Peter Marwedel. Scratchpad Overlay Approaches for Main/Scratchpad Memory Hierarchy. In *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*, pages 83–119. Springer, 2007.

[WA09] Jack Whitham and Neil Audsley. Implementing time-predictable load and store operations. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 265–274, 2009.

[WC92] Andrew Wolfe and Alex Chanin. Executing compressed programs on an embedded RISC architecture. 23(1-2):81–91, 1992. doi:10.1145/144965.145003.

[WDH+15] Peter Wägemann, Tobias Distler, Timo Hönig, Heiko Janker, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Worst-case energy consumption analysis for energy-constrained embedded systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 105–114. IEEE, 2015.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[WLL18] Yan Wang, Kenli Li, and Keqin Li. Dynamic Data Allocation and Task Scheduling on Multiprocessor Systems With NVM-Based SPM. *IEEE Access*, 7:1548–1559, 2018.

[WNNYE23] Simon Wegener, Kris K Nikov, Jose Nunez-Yanez, and Kerstin Eder. Energyanalyzer: Using static wcet analysis techniques to estimate the energy consumption of embedded applications. *arXiv preprint arXiv:2305.14968*, 2023.

[WXP10] Hui Wu, Jingling Xue, and Sri Parameswaran. Optimal WCET-aware code selection for scratchpad memory. In *Proceedings of the tenth ACM International Conference on Embedded Software*, pages 59–68, 2010.

[XLL07] Changjiu Xian, Yung-Hsiang Lu, and Zhiyuan Li. Energy-aware scheduling for real-time multiprocessor systems with uncertain task execution time. In *2007 44th ACM/IEEE Design Automation Conference*, pages 664–669. IEEE, 2007.

[XWL03] Yuan Xie, Wayne Wolf, and Haris Lekatsas. Profile-driven selective code compression [embedded systems]. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2003. doi:10.1109/date.2003.1253652.

[Yan12] Xin-She Yang. Flower Pollination Algorithm for Global Optimization. In *International Conference on Unconventional Computing and Natural Computation*, pages 240–249. Springer, 2012.

[YKH14] Xin-She Yang, Mehmet Karamanoglu, and Xingshi He. Flower pollination algorithm: a novel approach for multiobjective optimization. *Engineering Optimization*, 46(9):1222–1237, 2014.

[YWY+10] Yanqin Yang, Meng Wang, Haijin Yan, Zili Shao, and Minyi Guo. Dynamic scratch-pad memory management with data pipelining for embedded systems. *Concurrency and Computation: Practice and Experience*, 22(13):1874–1892, 2010.

[Zit99] Eckart Zitzler. *Evolutionary algorithms for multiobjective optimization: Methods and applications*, volume 63. Citeseer, 1999.

[ZKW+06] Wankang Zhao, William Kreahling, David Whalley, Christopher Healy, and Frank Mueller. Improving wcet by applying worst-case path optimizations. *Real-Time Systems*, 34:129–152, 2006.

[ZT99] Eckart Zitzler and Lothar Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.

Acronyms and Notational Conventions

Acronyms

DSPM	Data SPM.
ISPM	Instruction SPM.
`EEMBC`	Embedded Microprocessor Benchmark Consortium.
`EnergyAnalyzer`	Static Energy Consumption Analyzer.
`aiT`	Static WCET Analyzer.
ACEC	Average-Case Execution Count.
ACET	Average-Case Execution Time.
AHB	Advanced High-performance Bus.
ASIC	Application Specific Integrated Circuit.
BCET	Best-Case Execution Time.
CFG	Control Flow Graph.
CISC	Complex Instruction Set Computer.
CPU	Central Processing Unit.
DMA	Direct Memory Access.
DMS	Deadline-Monotonic Scheduling.
DRAM	Dynamic RAM.
EDF	Earliest Deadline First.
EEPROM	Electrically Erasable Programmable ROM.
FPA	Flower Pollination Algorithm.
ILP	Integer Linear Programming.
ISA	Instruction Set Architecture.
NVRAM	Non-Volatile RAM.
RAM	Random Access Memory.
RISC	Reduced Instruction Set Computer.
RMS	Rate-Monotonic Scheduling.
ROM	Read Only Memory.
SPEA	Strength Pareto Evolutionary Algorithm.
SPM	Scratchpad Memory.
SRAM	Static RAM.
WCC	WCET-Aware C Compiler.

WCEC	Worst-Case Execution Count.
WCECC	Worst-Case Energy Consumption Count.
WCECP	Worst-Case Energy Consumption Path.
WCEP	Worst-Case Execution Path.
WCET	Worst-Case Execution Time.

Acronyms for Optimizations

BL	A BaseLine evaluation where the whole program is executed from the Flash memory.
DCPO	DMA Call Placement Optimization.
ICA	Imbact-based Constraint Approach to perform multi-objective static SPM allocation optimization for multi-task systems.
MO_{Approx}	Approximation-aware multi-objective static SPM allocation optimization.
MO_S	Multi-objective static SPM allocation optimization.
MTMO	Multi-objective static SPM allocation optimization for multi-task systems.
MO_C	Compression-aware multi-objective dynamic SPM allocation optimization.
MO_{DMA}	DMA-aware multi-objective dynamic SPM allocation optimization.
MO_D	Multi-objective dynamic SPM allocation optimization.
PCA	Path-based Constraint Approach to perform multi-objective static SPM allocation optimization for multi-task systems.
SO_S	Single-objective ILP-based static SPM allocation optimization.
SO_F	Function-level single-objective ILP-based static compression-aware optimization.
SO_{DMA}	DMA-aware single-objective ILP-based dynamic SPM allocation optimization.
SO_D	Single-objective ILP-based dynamic SPM allocation optimization.

Symbols Used for Optimizations

X	Search or decision space.
x	Solution vector.
d	Dimension of the search space X.
Y	Objective space.
$F(.)$	Objective function vector.
$F_k(.)$	k^{th} objective function.

$Q_c(.)$	c^{th} constraint applied on the objective function.
PF	Pareto Optimal Front.
PF_F	Pareto Optimal Front obtained from the current evaluation run.
PF_T	Final Pareto Optimal Front obtained from all the optimization runs.

Metaheuristic Algorithms

$\mathcal{C}$	Coverage.
$\mathtt{NR}$	Non-dominance Ratio.
$\mathtt{HV}$	HyperVolume.
$\mathtt{RNS}$	Ratio of Non-dominated Solutions.
N	Size of the Population.
$maxGen$	Maximum number of generations.
$\mathbf{P}^g$	A population at the g^{th} generation.
$x^{(g,n)}$	n^{th} decision (solution) vector x in the population $\mathbf{P}^g$ of the g^{th} generation of the metaheuristic algorithm.
$\tilde{\mathbf{P}}^g$	External population at the g^{th} generation for SPEA.
p_c	Crossover probability for SPEA.
p_m	Mutation probability for SPEA.
λ	Positive constant integer for the standard gamma function in the FPA algorithm.
$\Phi(.)$	Pollination operator for FPA.
$\Gamma(\lambda)$	Standard gamma function for FPA.
γ	Scaling factor for FPA.

Static Allocation

x	Binary decision variable vector for the static allocation problem that decides if basic blocks and global data objects will be statically allocated to the SPM or not.
$\mathcal{G}_g$	g^{th} global data object within the program.
B_i	Code size of the basic block b_i.
Ψ_J	Extra cost incurred due to jump correction in terms of code size.
$S_{\mathcal{G}_g}$	Size of the global data object $\mathcal{G}_g$.
S_{ISPM}	Size of the ISPM.
S_{DSPM}	Size of the DSPM.
$\tilde{F}(.)$	Approximated objective function vector.

$W_{(f,i)}$	WCET value of basic block b_i for individual f.
$E_{(f,i)}$	Energy consumption value of basic block b_i for individual f.
τ_t	A t^{th} task in a multi-task system.
Γ	Set of all tasks of a multi-task system.
Γ_r	Set of tasks removed from Γ such that the multi-task system of remaining tasks is schedulable.
T	Total number of tasks in a multi-task system.
W_t and E_t	WCET and energy values of the t^{th} task.
$\mathbb{W}^{(g,n)}$ and $\mathbb{E}^{(g,n)}$	Set of basic blocks on WCEP and WCECP of the n^{th} individual of the g^{th} generation, respectively.
$w_{t_i}^{(g,n)}$ and $e_{t_i}^{(g,n)}$	WCEC and WCECC of the basic block b_i in the t^{th} task for the n^{th} individual of the g^{th} generation, respectively.
$\mathbb{M}_{t_i}$	Impact metric for basic block b_i in the t^{th} task.

Dynamic Allocation

$\mathcal{M}_{(p,m)}$	m^{th} memory object in the p^{th} function.
M_p	Total number of memory objects in the p^{th} function.
$\mathtt{MO}$	Set of memory objects within the program.
x	Binary decision variable vector for the dynamic allocation problem that decides if memory objects and global data objects are dynamically and statically placed in ISPM and DSPM or not, respectively.
$\mathcal{A}_{(p,m,a)}$	a^{th} address object in the m^{th} memory object of the p^{th} function.
$A_{(p,m)}$	Total number of address objects in the m^{th} memory object of the p^{th} function.
$\mathbf{A}$	Number of address objects that need to be dynamically allocated according to the solution vector x.
ϵ	Total number of address objects assigned to SPM after solving the address assignment problem.
$\mathcal{L}_{(p,m)}$	Set of basic blocks that define the live range of the m^{th} memory object of the p^{th} function.
$\alpha_{(p,m,a)}$	Start address of the address object $\mathcal{A}_{(p,m,a)}$ in the Flash memory.
$\beta_{(p,m,a)}$	Size of the address object $\mathcal{A}_{(p,m,a)}$.
$\gamma_{(p,m,a)}$	Destination address of the address object $\mathcal{A}_{(p,m,a)}$ in SPM.
b_p^r	r^{th} basic block in the p^{th} function.
B_p	Total number of basic blocks in the p^{th} function.
α_p^r	Start address in the Flash memory of the r^{th} basic block in the p^{th} function.

β_p^r	Size of the r^{th} basic block in the p^{th} function.
$S_{\mathcal{M}_{(p,m)}}$	Size of the memory object $\mathcal{M}_{(p,m)}$.

DMA Call Placement Optimization

aT_F	Access latency for the Flash memory.
aT_S	Access latency for the SPM.
$t_s(\beta)$	Cost of loading β bytes from Flash to SPM using DMA in burst mode.
t_Ω	Total number of cycles required to perform arbitration once.
$\mathcal{T}(x)$	Total execution time needed for DMA transfer in burst mode for solution x.
$\delta_{(p,m,a)}$	DMA transfer call for address object $\mathcal{A}_{(p,m,a)}$.
y	Binary decision variable vector that decides if a DMA call is placed after a particular basic block.
$\mathcal{T}'(y)$	Total execution time needed for DMA transfer in arbitration mode for solution vector y.
$\sigma^r_{(p,m,a)}$	DMA call placement cost if $\delta_{(p,m,a)}$ is placed after basic block b_p^r.
$\nu^q_{(p,m)}$	Binary decision variable that is 1 if basic block b_p^q is a *def* basic block of $\mathcal{M}_{(p,m)}$.
$t^j_{(p,m,a)}$	Basic block-level placement cost for basic block b_p^j.
$\eta^j_{(p,m,a)}$	Binary constant that indicates if the $t^j_{(p,m,a)}$ for basic block b_p^j will be considered or not while calculating the total execution time cost.
$\mathbb{F}_p^r$	Execution frequency of the basic block b_p^r.
$\mathbf{N}_l^j$	Total number of load instructions in the j^{th} basic block that access memory.
$\mathbf{N}_s^j$	Total number of store instructions in the j^{th} basic block that access memory.
$\mathbf{N}^j$	Total number of instructions in the j^{th} basic block.
$\mathbf{N}_\Omega^j$	Maximum number of times arbitration can occur in the j^{th} basic block.
t_R^j	Relative placement cost for the j^{th} basic block.

Compression

$\mathbb{CO}_{(p,m,c)}$	c^{th} compression object in the m^{th} memory object of the p^{th} function.

$C_{(p,m)}$	Total number of compression objects in the m^{th} memory object of the p^{th} function.
$\mathbf{C}$	Number of compression objects that need to be compressed and dynamically decompressed according to the solution vector x.
CR	Compression ratio.
$\mathbb{F}$	Map of between a function (f_p) and the list of functions that it is dependent on.
$\mathcal{CF}_p$	List of functions on which the function f_p depends, i.e., f_p calls functions in this list.
$\mathbb{S}$	Ordered vector of functions in which function compression must occur.
CS^F	Size of the code in the Flash memory.
CS^D	Size of the compressed objects in the data section.
CS^C	Size of the compressed version of the code, i.e., $CS^C = CS^F + CS^D$.
CS^S	Size of the decompressed code when placed in SPM.
CS^{UC}	Size of the static uncompressed version of the code, i.e., $CS^{UC} = CS^F + CS^S$.

List of Figures

2.1. Exemplary distribution of all possible execution times of a program and distribution of measured execution times adapted from Ermedahl [Erm03] and Wilhelm et al. [WEE+08]. 11
2.2. Exemplary von Neumann architecture 17
2.3. Exemplary Harvard architecture 17
2.4. ARM Cortex-M0 architecture 21

3.1. WCC infrastructure: The optimization and analysis parts, represented in green, differentiate the WCC compiler from general modern compilers. 25
3.2. Exemplary CFG with two functions `main()` and `foo()` 29
3.3. Basic block assignment for the exemplary CFG from Figure 3.2, where `C` and `F` are placed in SPM 30
3.4. Exemplary code transformations for SPM allocation 31
3.5. WCET and energy analyses framework 33

4.1. Exemplary Pareto front for a minimization problem. Let F_1 and F_2 be two objective functions that can contradict each other. The objective space for the minimization problem is 2-dimensional, and the objective values for the solutions are plotted in this objective space. The non-dominated and dominated points in the objective space are represented by green pentagons and red dots, respectively. The Pareto front, represented by a green dotted line, is formed by joining all non-dominated points. 41
4.2. Exemplary Pareto fronts to explain quality indicators for a minimization problem, where F_1 and F_2 are two contradicting objectives 45

5.1. An exemplary architecture for static SPM allocation 59
5.2. Multi-objective static SPM allocation optimization framework . . 61
5.3. Pareto fronts for four benchmarks using MO_S, SO_S, and BL . . . 67
5.4. Compilation times for MO_S and SO_S optimization runs 71

6.1. An example with four possible allocation cases covered by the individuals. We consider only five basic blocks, `A`, `B`, `C`, `D`, and `E`, in this example, where `A` has two successors (`B` and `C`). Let the decisions for the five basic blocks be arranged alphabetically as (`A`, `B`, `C`, `D`, `E`) in the decision vector x. 79

6.2. Two exemplary allocations for the CFG from Figure 6.1 that are not covered by the four individuals $\mathbb{A}, \mathbb{B}, \mathbb{C}$, and $\mathbb{D}$ 80
6.3. Multi-objective static SPM allocation with the approximation model for approximating WCET and energy objectives 87
6.4. Compilation times for MO_{Approx}, MO_S, and SO_S optimizations . 90
6.5. Percentage decrease in the compilation times achieved by using approximation-based multi-objective static SPM allocation 91
6.6. Pareto fronts for four benchmarks using MO_S, SO_S, BL, and MO_{Approx} 92

7.1. Compilation times required to perform MTMO, PCA, and ICA for task sets with 2 and 4 tasks, respectively 112
7.2. Compilation times required to perform MTMO, PCA, and ICA for task sets with 6 and 8 tasks, respectively 113
7.3. Pareto fronts obtained using MTMO, PCA, ICA, and BL 116

8.1. An exemplary architecture for dynamic SPM allocation 125
8.2. CFG of the exemplary program code from Listing 8.1 131
8.3. A nested loop example to show the difference between memory objects and address objects 133
8.4. Multi-objective dynamic SPM allocation framework 135
8.5. CFG of an exemplary program code with loop nests 138
8.6. Pareto fronts for four benchmarks obtained using MO_S, MO_D, SO_D, and BL evaluation runs 149
8.7. Compilation times for MO_S, MO_D, and SO_D optimization runs . 155

9.1. Exemplary CFG with DMA transfer call 158
9.2. Exemplary memory hierarchy with DMA 159
9.3. DMA-aware multi-objective dynamic SPM allocation 165
9.4. CFG of the exemplary program code from Listing 8.1 170
9.5. Exemplary DMA call placements with arbitration 171
9.6. Exemplary consecutive DMA call placements 171
9.7. Exemplary DMA call placements in burst mode 172
9.8. Exemplary CFG with DMA transfer call for DMA placement cost calculation: We assume that $\forall j = \overline{1,4}$, $r = 1$, and arbitration time $t_\Omega = 2$ cycles. 176
9.9. Exemplary DMA placement cost calculation: We assume that $\forall j = \overline{1,4}$, $r = 1$, arbitration time $t_\Omega = 2$ cycles, and burst mode DMA call cost $t_s(\beta_{(p,m,a)}) = 40$ cycles 177
9.10. Exemplary DMA placement cost calculation: We assume that $\forall j = \overline{1,4}$, $r = 1$, arbitration time $t_\Omega = 2$ cycles, and burst mode DMA call cost $t_s(\beta_{(p,m,a)}) = 120$ cycles 178
9.11. Exemplary DMA placement cost calculation: We assume that $\forall j = \overline{1,4}$, $r = 1$, arbitration time $t_\Omega = 2$ cycles, and burst mode DMA call cost $t_s(\beta_{(p,m,a)}) = 200$ cycles 178

9.12. Comparison between the relative execution metrics (T_κ) for FPA-, SPEA-, and ILP-based DCPO 191
9.13. Pareto front for four benchmarks with 60% relative SPM size and MO_{DMA}+DCPO–FPA configuration 194
9.14. Compilation times for MO_S, MO_D, SO_D, SO_{DMA}, and MO_{DMA}+DCPO–FPA runs with 60% SPM size configuration . . 199
9.15. Pareto front for four benchmarks with 60% relative SPM size and MO_{DMA}+DCPO–ILP configuration 201
9.16. Compilation times for MO_S, MO_D, SO_D, SO_{DMA}, and MO_{DMA}+DCPO–ILP runs with 60% SPM size configuration . . . 203

10.1. Exemplary architecture for compression-aware multi-objective dynamic SPM allocation optimization 212
10.2. Exemplary original code before compression. We assume that the loop L_1, the function `f1()`, and the function `f2()` will be dynamically decompressed during runtime. 218
10.3. Exemplary code transformations needed before generating dynamically allocatable code for the example from Figure 10.2 . . . 219
10.4. Exemplary compression-aware code at compile time 220
10.5. Exemplary compression-aware code at runtime 220
10.6. Compression-aware multi-objective dynamic SPM allocation framework 222
10.7. Pareto fronts for four benchmarks using MO_D, BL, and MO_C . . 233
10.8. Increase in code size comparison (in %) between MO_D code size and compressed and uncompressed versions of the MO_C code relative to the BL code (0% is the BL code size) 236
10.9. Compilation times for MO_D and MO_C optimization runs 239

A.1. Pareto fronts for the remaining benchmarks using MO_S, SO_S, and BL approaches 285
A.2. (Continued) Pareto fronts for the remaining benchmarks 286
A.3. (Continued) Pareto fronts for the remaining benchmarks 287
A.4. (Continued) Pareto fronts for the remaining benchmarks 288

B.1. Pareto fronts for the remaining benchmarks using MO_S, SO_S, BL, and MO_{Approx} 291
B.2. (Continued) Pareto fronts for the remaining benchmarks 292
B.3. (Continued) Pareto fronts for the remaining benchmarks 293

C.1. Pareto fronts for the remaining task sets using MTMO, PCA, ICA, and BL evaluation runs for multi-task systems 299
C.2. (Continued) Pareto fronts for the remaining task sets 300
C.3. (Continued) Pareto fronts for the remaining task sets 301
C.4. (Continued) Pareto fronts for the remaining task sets 302
C.5. (Continued) Pareto fronts for the remaining task sets 303

D.1. Pareto fronts for the remaining benchmarks using MO_D, MO_S, SO_D, and BL evaluation runs 307
D.2. (Continued) Pareto fronts for the remaining benchmarks 308

E.1. Pareto fronts for the remaining benchmarks using MO_D, MO_S, SO_S, SO_{DMA}, BL, and MO_{DMA} optimization approaches for the relative 60% SPM size and DCPO–FPA case 315
E.2. (Continued) Pareto fronts for the remaining benchmarks for the relative 60% SPM size and DCPO–FPA case 316
E.3. Pareto fronts for the remaining benchmarks using MO_D, MO_S, SO_S, SO_{DMA}, BL, and MO_{DMA} optimization approaches for the relative 60% SPM size and DCPO–ILP case 317
E.4. (Continued) Pareto fronts for the remaining benchmarks for the relative 60% SPM size and DCPO–ILP case 318
E.5. Pareto fronts for all the benchmarks using MO_D, MO_S, SO_S, SO_{DMA}, BL, and MO_{DMA} optimization approaches for the relative 20% SPM size and DCPO–FPA case 319
E.6. (Continued) Pareto fronts for all the benchmarks for the relative 20% SPM size and DCPO–FPA case 320
E.7. (Continued) Pareto fronts for all the benchmarks for the relative 20% SPM size and DCPO–FPA case 321
E.8. Pareto fronts for all the benchmarks using MO_D, MO_S, SO_S, SO_{DMA}, BL, and MO_{DMA} optimization approaches for the relative 40% SPM size and DCPO–FPA case 322
E.9. (Continued) Pareto fronts for all the benchmarks for the relative 40% SPM size and DCPO–FPA case 323
E.10. (Continued) Pareto fronts for all the benchmarks for the relative 40% SPM size and DCPO–FPA case 324
E.11. Pareto fronts for all the benchmarks using MO_D, MO_S, SO_S, SO_{DMA}, BL, and MO_{DMA} optimization approaches for the relative 80% SPM size and DCPO–FPA case 325
E.12. (Continued) Pareto fronts for all the benchmarks for the relative 80% SPM size and DCPO–FPA case 326
E.13. (Continued) Pareto fronts for all the benchmarks for the relative 80% SPM size and DCPO–FPA case 327

F.1. Exemplary basic block code with three instructions, where the basic block is in the Flash memory. We assume that the basic block has one load and one store instruction, and the access latency for the Flash memory is 3 cycles ($aT_F = 3$). 330
F.2. Exemplary basic block code with three instructions, where the basic block is in SPM. We assume that the basic block has one load and one store instruction, and the access latency for the SPM is 1 cycle ($aT_S = 1$). 331

F.3. Example basic block from Figure F.1, when the DMA controller and the CPU are working in parallel. 333
F.4. Example basic block from Figure F.2, when the DMA controller and CPU are working in parallel. 334

G.1. Pareto fronts for the remaining benchmarks using MO_D, BL, and MO_C optimization approaches 343
G.2. (Continued) Pareto fronts for the remaining benchmarks 344

List of Tables

5.1. Values for the FPA algorithm's hyperparameters 65
5.2. Values for the SPEA algorithm's hyperparameters 65
5.3. Comparison between the average number of solutions (in %) on the final PF_T from the respective PF_F of each optimization approach 68
5.4. Quality Indicators for MO_S, SO_S, and BL optimization approaches 69
5.5. Total number of benchmarks with best (or nondominated) quality indicators for each optimization approach 70

6.1. Comparison between the average number of solutions (in %) on the final PF_T from the respective PF_F of each optimization approach 93
6.2. Total number of benchmarks with best (or nondominated) quality indicators for each optimization approach 95

7.1. Comparison between the average number of solutions (in %) on the final PF_T from the respective PF_F of each optimization approach 117
7.2. Average reduction in WCET, energy consumption, and schedulability (in %) obtained using MTMO, PCA, and ICA over BL . . . 117
7.3. Total number of benchmarks with best (or non-dominated) quality indicators for each optimization approach 118

8.1. Comparison between the average number of solutions (in %) on the final PF_T from the respective PF_F of each optimization approach 151
8.2. Average reduction in WCET and energy consumption objectives (in %) obtained using MO_S, MO_D, and SO_D over BL 152
8.3. Total number of benchmarks with best (or nondominated) quality indicators for each optimization approach 153

9.1. Comparison between the average number of solutions (in %) on the final PF_T from respective PF_F of each optimization approach for 60% relative SPM size and MO_{DMA}+DCPO–FPA configuration 195
9.2. Average reduction in WCET and energy consumption objectives (in %) obtained using MO_S, MO_D, SO_D, SO_{DMA}, and MO_{DMA}+DCPO–FPA over the BL evaluation run 196
9.3. Total number of benchmarks with best (or nondominated) quality indicators for each optimization approach for 60% relative SPM size and MO_{DMA}+DCPO–FPA configuration 197

9.4. Comparison between the average number of solutions (in %) on the final PF_T from respective PF_F of each optimization approach for 60% relative SPM size and MO_{DMA}+DCPO–ILP configuration 200
9.5. Average reduction in WCET and energy consumption objectives (in %) obtained using MO_S, MO_D, SO_D, SO_{DMA}, and MO_{DMA}+DCPO–ILP over the BL evaluation run 201
9.6. Total number of benchmarks with best (or nondominated) quality indicators for each optimization approach for 60% relative SPM size and MO_{DMA}+DCPO–ILP configuration 202
9.7. Reduction of average WCET and energy consumption (in %) obtained using MO_S, MO_D, SO_D, SO_{DMA}, and MO_{DMA}+DCPO–FPA relative to BL, for 20%, 40%, and 80% relative SPM size configuration 205
9.8. Average per-benchmark reduction in WCET and energy consumption (in %) obtained using MO_{DMA} over MO_D, for 20%, 40%, and 80% relative SPM size configuration 205
9.9. Total number of benchmarks with best (or nondominated) quality indicators for 20%, 40%, and 80% relative SPM size configuration 206
9.10. Comparison between the average number of solutions (in %) on the final PF_T from respective PF_F of each optimization approach for 20%, 40%, and 80% relative SPM size configuration 207

10.1. Comparison between the average number of solutions (in %) on the final PF_T from the respective PF_F of each optimization approach 234
10.2. Average reduction in WCET and energy consumption objectives obtained using MO_D and MO_C over the BL evaluation run . . . 234
10.3. Average increase in code size due to MO_D and MO_C over BL . . 236
10.4. Total number of benchmarks with best (or nondominated) quality indicators for each optimization approach 238

B.1. Coverage ($\mathcal{C}$) and Non-dominance Ratio (NR) metrics for MO_S, SO_S, BL, and MO_{Approx} optimization approaches 289
B.2. Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for MO_S, SO_S, BL, and MO_{Approx} optimization approaches 290

C.1. Characterization of the evaluated task sets 296
C.2. Coverage ($\mathcal{C}$) and Non-dominance Ratio (NR) metrics for MTMO, PCA, ICA, and BL evaluations 297
C.3. Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for MTMO, PCA, ICA, and BL evaluations 298

D.1. Coverage ($\mathcal{C}$) and Non-dominance Ratio (NR) for MO_D, MO_S, SO_D, and BL evaluation runs 305
D.2. Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) for MO_D, MO_S, SO_D, and BL evaluation runs 306

E.1. Coverage ($\mathcal{C}$) and Non-dominance Ratio (NR) metrics for relative 60% SPM size and DCPO–FPA 310
E.2. Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for relative 60% SPM size and DCPO–FPA 310
E.3. Coverage ($\mathcal{C}$) and Non-dominance Ratio (NR) metrics for relative 60% SPM size and DCPO–ILP 311
E.4. Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for relative 60% SPM size and DCPO–ILP 311
E.5. Coverage ($\mathcal{C}$) and Non-dominance Ratio (NR) metrics for relative 20% SPM size and DCPO–FPA 312
E.6. Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for relative 20% SPM size and DCPO–FPA 312
E.7. Coverage ($\mathcal{C}$) and Non-dominance Ratio (NR) metrics for relative 40% SPM size and DCPO–FPA 313
E.8. Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for relative 40% SPM size and DCPO–FPA 313
E.9. Coverage ($\mathcal{C}$) and Non-dominance Ratio (NR) metrics for relative 80% SPM size and DCPO–FPA 314
E.10. Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for relative 80% SPM size and DCPO–FPA 314

G.1. Coverage ($\mathcal{C}$) and Non-dominance Ratio (NR) metrics for MO_D, BL, and MO_C evaluation runs 341
G.2. Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for MO_D, BL, and MO_C evaluation runs 342

Listings

3.1. Exemplary Memory Specification within WCC 26
3.2. Exemplary loop bound annotation 28
3.3. Exemplary flow restriction annotation 29
3.4. Exemplary original function call code 30
3.5. Exemplary function call code after jump correction 31
3.6. Exemplary original fall-through code 32
3.7. Exemplary fall-through code after jump correction 32

8.1. Exemplary program code 130

Appendices

A

Multi-Objective Static SPM Allocation

Figure A.1, Figure A.2, Figure A.3, and Figure A.4 present the Pareto fronts for all the remaining benchmarks from the evaluations described in Chapter 5 for MO_S, SO_S, and BL evaluation runs.

Figure A.1.: Pareto fronts for the remaining benchmarks using MO_S, SO_S, and BL approaches

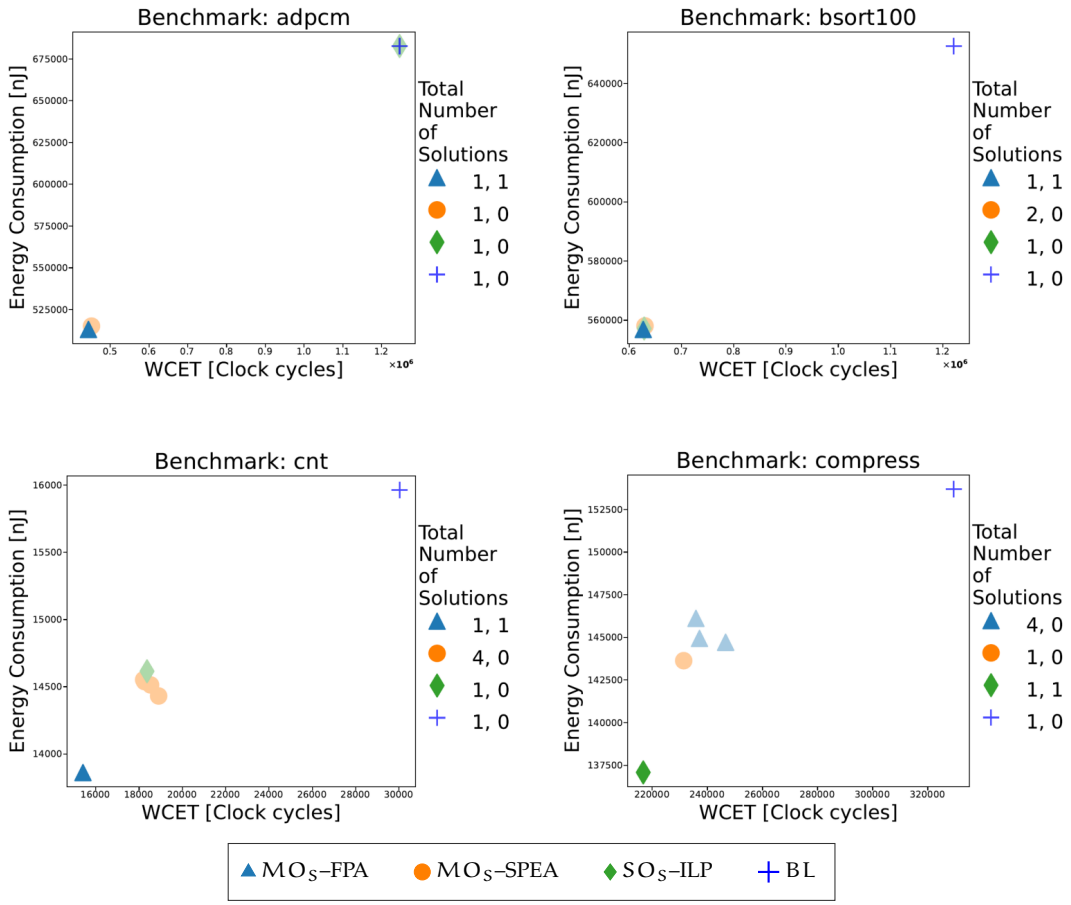


Figure A.2.: (Continued) Pareto fronts for the remaining benchmarks

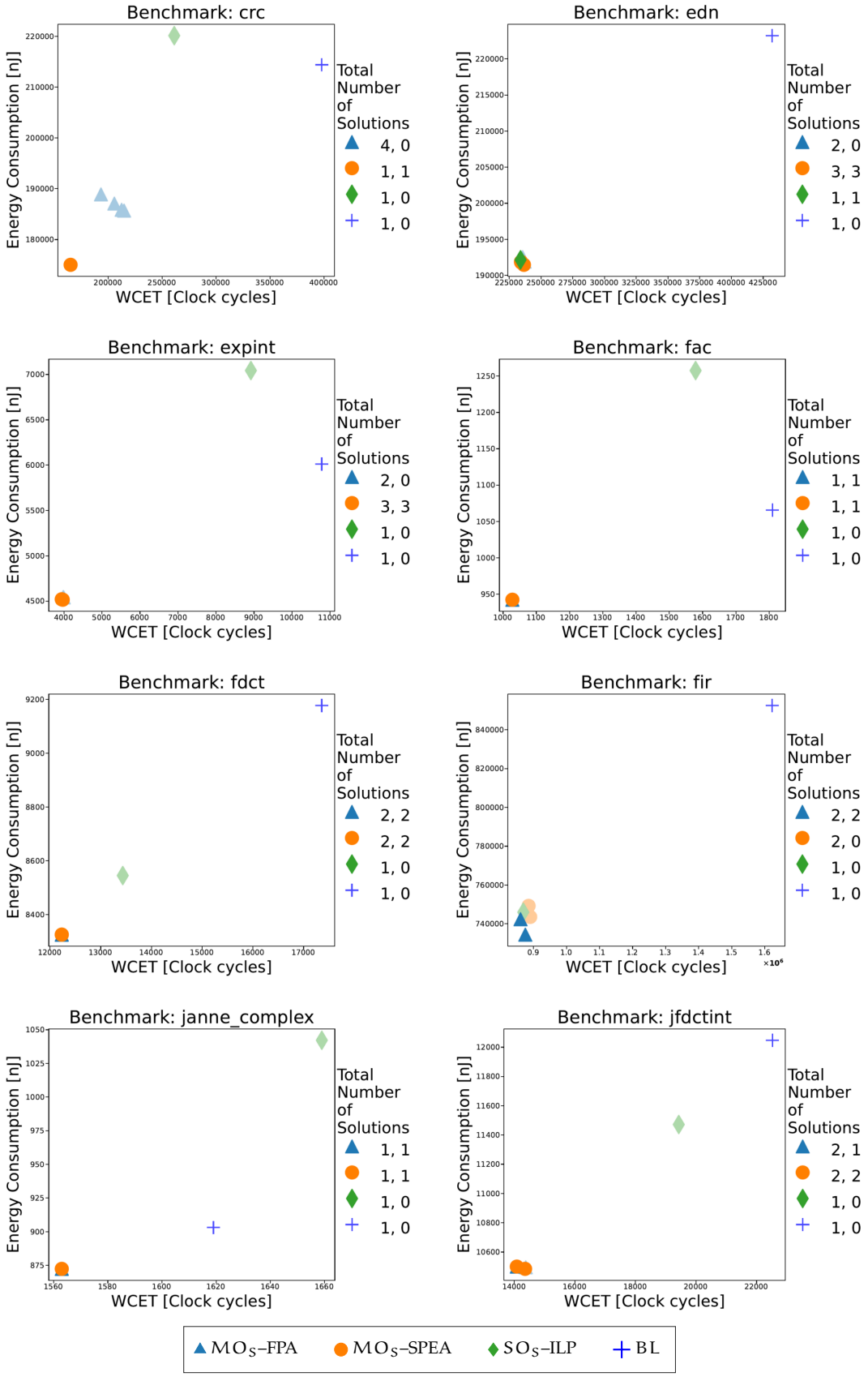


Figure A.3.: (Continued) Pareto fronts for the remaining benchmarks

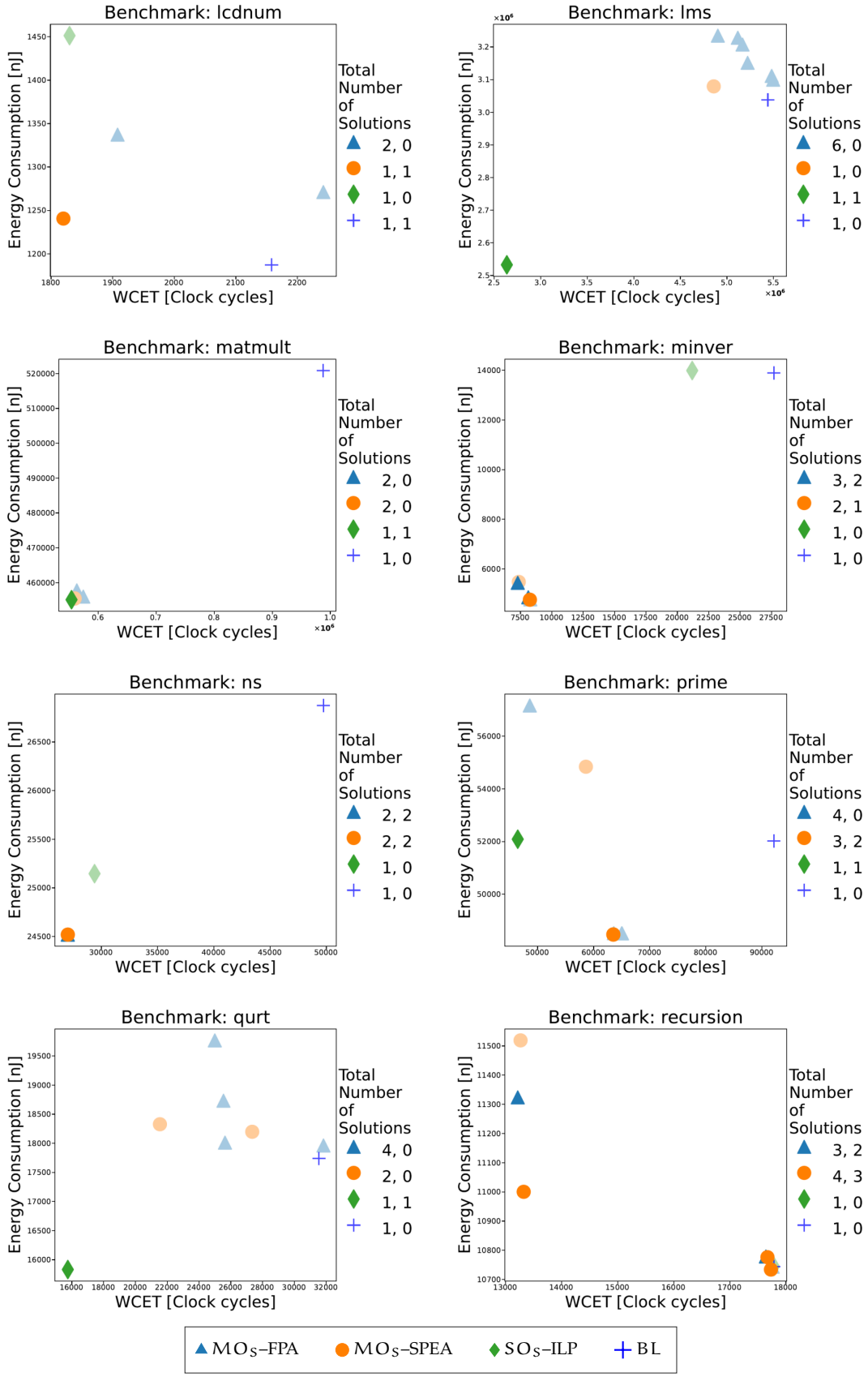
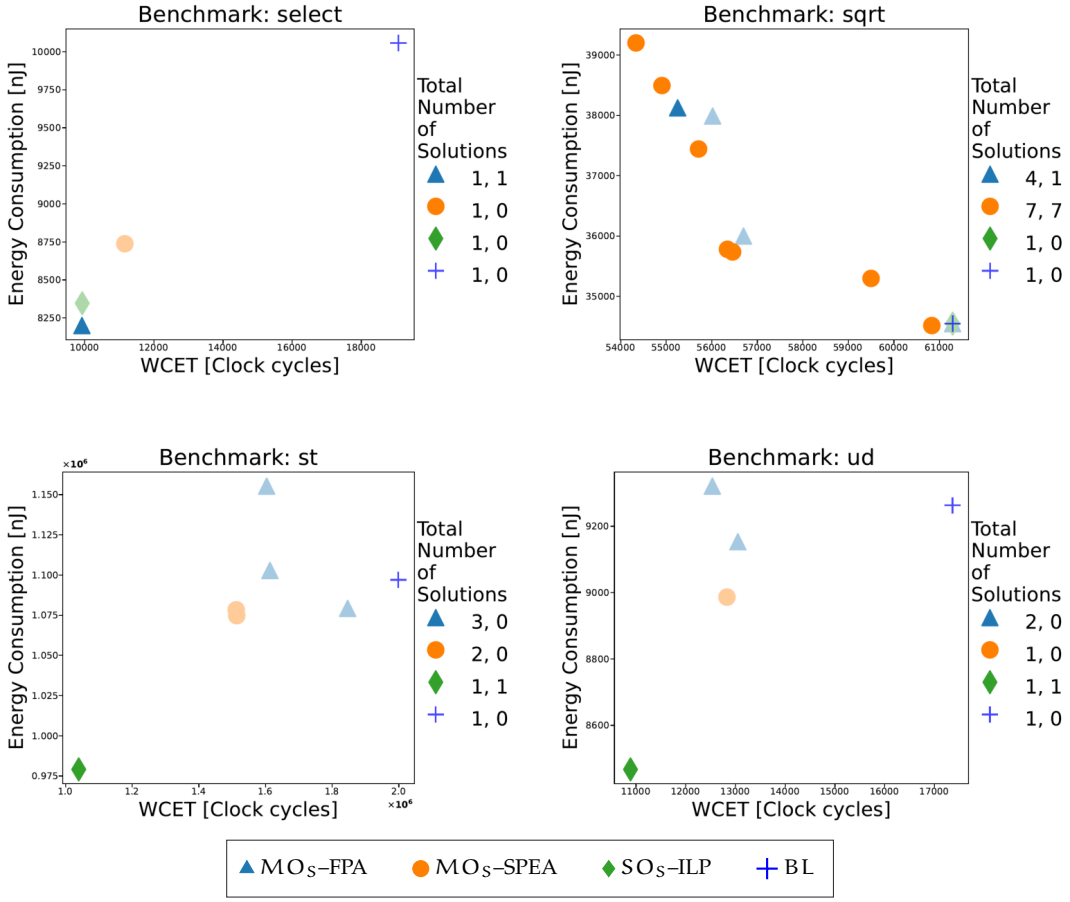


Figure A.4.: (Continued) Pareto fronts for the remaining benchmarks



B

Approximating WCET and Energy for Fast Multi-Objective Static SPM Allocation

Tables B.1 and B.2 present the quality indicator values for all benchmarks from the evaluations described in Chapter 6. For any benchmark, the approach with quality indicators in bold performed better (or nondominant) than other approaches. Figure B.1, Figure B.2, and Figure B.3 present the Pareto fronts for all the remaining benchmarks from the evaluations described in Chapter 6.

Table B.1.: Coverage ($\mathcal{C}$) and Non-dominance Ratio (NR) metrics for MO_S, SO_S, BL, and MO_{Approx} optimization approaches

	Coverage ($\mathcal{C}$)						Non-dominated Ratio (NR)					
	MO_S		SO_S	BL	MO_{Approx}		MO_S		SO_S	BL	MO_{Approx}	
Benchmarks	FPA	SPEA	ILP		FPA	SPEA	FPA	SPEA	ILP		FPA	SPEA
adpcm	1	1	1	1	1	**0**	0	0	0	0	0	**1**
bs	1	1	1	1	**0**	**0**	0	0	0	0	**0.571**	0.429
bsort100	**0**	1	1	1	1	**0**	**0.5**	0	0	0	0	**0.5**
cnt	**0**	1	1	1	1	**0**	0.333	0	0	0	0	**0.667**
compress	1	1	**0**	1	1	1	0	0	**1**	0	0	0
crc	1	1	1	1	**0**	**0**	0	0	0	0	**0.5**	**0.5**
edn	1	1	**0**	1	**0**	1	0	0	**0.5**	0	**0.5**	0
expint	1	**0**	1	1	1	1	0	**1**	0	0	0	0
fac	1	1	1	1	**0**	**0**	0	0	0	0	**0.5**	**0.5**
fdct	1	1	1	1	0.5	**0**	0	0	0	0	0.143	**0.857**
fibcall	**0**	**0**	**0**	**0**	**0**	**0**	**0.167**	**0.167**	**0.167**	**0.167**	**0.167**	**0.167**
fir	0.5	1	1	1	1	**0**	**0.5**	0	0	0	0	**0.5**
insertsort	1	1	1	1	**0**	**0**	0	0	0	0	**0.5**	**0.5**
janne_complex	1	1	1	1	**0**	**0**	0	0	0	0	**0.5**	**0.5**
jfdctint	0.5	**0**	1	1	0.667	1	0.25	**0.5**	0	0	0.25	0
lcdnum	1	1	1	1	0.5	**0**	0	0	0	0	0.333	**0.667**
lms	1	1	**0**	1	1	1	0	0	**1**	0	0	0
matmult	1	1	**0**	1	0.75	1	0	0	**0.5**	0	**0.5**	0
minver	**0.333**	0.5	1	1	1	1	**0.667**	0.333	0	0	0	0
ns	1	1	1	1	**0**	**0**	0	0	0	0	**0.667**	0.333
prime	1	0.333	1	1	1	**0**	0	**0.5**	0	0	0	**0.5**
qsort-exam	1	1	**0**	1	1	1	0	0	**1**	0	0	0
qurt	1	1	**0**	1	1	1	0	0	**1**	0	0	0
recursion	1	1	1	1	1	**0**	0	0	0	0	0	**1**
select	**0**	1	1	1	1	1	**1**	0	0	0	0	0
sqrt	1	1	1	1	0.5	**0**	0	0	0	0	0.333	**0.667**
st	1	1	**0**	1	1	1	0	0	**1**	0	0	0
ud	1	1	**0**	1	1	1	0	0	**1**	0	0	0

Table B.2.: Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for MO_S, SO_S, BL, and MO_{Approx} optimization approaches

	Ratio of Non-dominated Solutions (RNS)						HyperVolume (HV)					
	MO_S		SO_S	BL	MO_{Approx}		MO_S		SO_S	BL	MO_{Approx}	
	FPA	SPEA	ILP		FPA	SPEA	FPA	SPEA	ILP		FPA	SPEA
Benchmarks												
adpcm	0	0	0	0	0	**1**	0.617	0.604	0	0	0.992	**0.998**
bs	0	0	0	0	**1**	**1**	0.387	0.387	0.002	0.359	**0.57**	**0.57**
bsort100	**1**	0	0	0	0	**1**	**0.997**	0.979	0.991	0	0.576	0.983
cnt	**1**	0	0	0	0	**1**	0.824	0.492	0.429	0	0.856	**0.907**
compress	0	0	**1**	0	0	0	0.444	0.523	**0.991**	0	0.719	0.706
crc	0	0	0	0	**1**	**1**	0.441	0.661	0	0	0.985	**0.987**
edn	0	0	**1**	0	**1**	0	0.972	0.993	0.973	0	**0.995**	0.986
expint	0	**1**	0	0	0	0	0.873	**0.887**	0	0.046	0.759	0.662
fac	0	0	0	0	**1**	**1**	0.566	0.566	0.001	0.194	**0.636**	**0.636**
fdct	0	0	0	0	0.5	**1**	0.840	0.840	0.501	0	0.859	**0.86**
fibcall	**1**	**1**	**1**	**1**	**1**	**1**	**0.492**	**0.492**	0.005	**0.492**	**0.492**	**0.492**
fir	0.5	0	0	0	0	**1**	0.975	0.872	0.871	0	0.962	**0.984**
insertsort	0	0	0	0	**1**	**1**	0.78	0.78	0.762	0	**0.82**	**0.82**
janne_complex	0	0	0	0	**1**	**1**	0.195	0.195	0.001	0.151	**0.604**	**0.604**
jfdctint	0.5	**1**	0	0	0.333	0	0.895	**0.901**	0.144	0	0.872	0.87
lcdnum	0	0	0	0	0.5	**1**	0.145	0.198	0.001	0.189	0.62	**0.621**
lms	0	0	**1**	0	0	0	0.013	0.049	**1**	0.006	0.031	0.059
matmult	0	0	**1**	0	0.25	0	0.928	0.946	0.962	0	**0.989**	0.98
minver	**0.667**	0.5	0	0	0	0	**0.952**	0.949	0.071	0.01	0.021	0.024
ns	0	0	0	0	**1**	**1**	0.714	0.714	0.472	0	**0.96**	**0.96**
prime	0	0.667	0	0	0	**1**	0.588	0.619	0.545	0.012	0.602	**0.927**
qsort-exam	0	0	**1**	0	0	0	0.887	0.966	**0.985**	0	0.459	0.436
qurt	0	0	**1**	0	0	0	0.18	0.239	**0.944**	0.037	0.341	0.283
recursion	0	0	0	0	0	**1**	0.06	0.107	0.026	0.026	0.709	**0.896**
select	**1**	0	0	0	0	0	**0.91**	0.568	0.837	0	0.547	0.597
sqrt	0	0	0	0	0.5	**1**	0.353	0.414	0.078	0.078	**0.778**	0.739
st	0	0	**1**	0	0	0	0.141	0.232	**0.999**	0	0.146	0.298
ud	0	0	**1**	0	0	0	0.124	0.256	**0.881**	0.008	0.268	0.359

Figure B.1.: Pareto fronts for the remaining benchmarks using MO_S, SO_S, BL, and MO_{Approx}

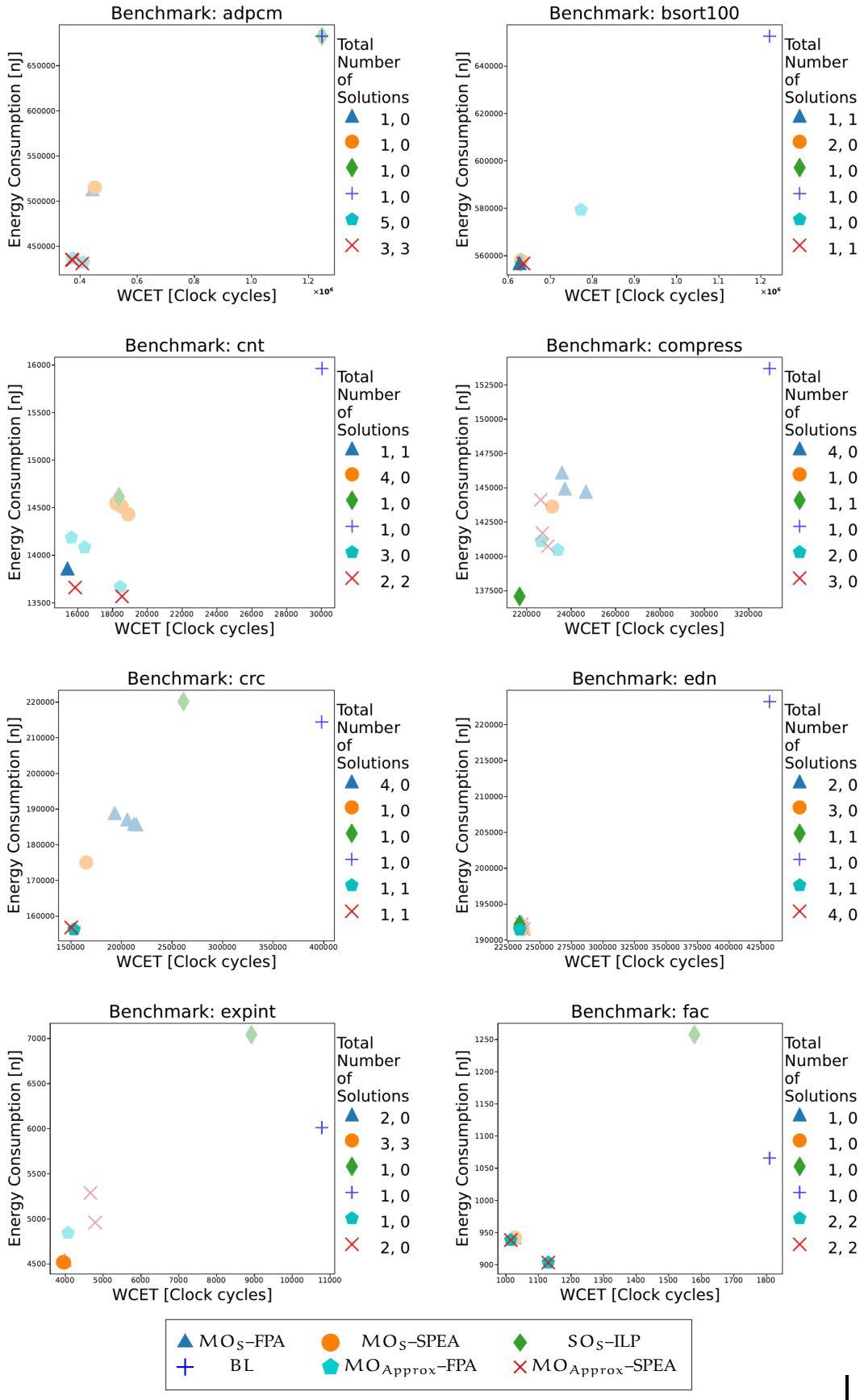


Figure B.2.: (Continued) Pareto fronts for the remaining benchmarks

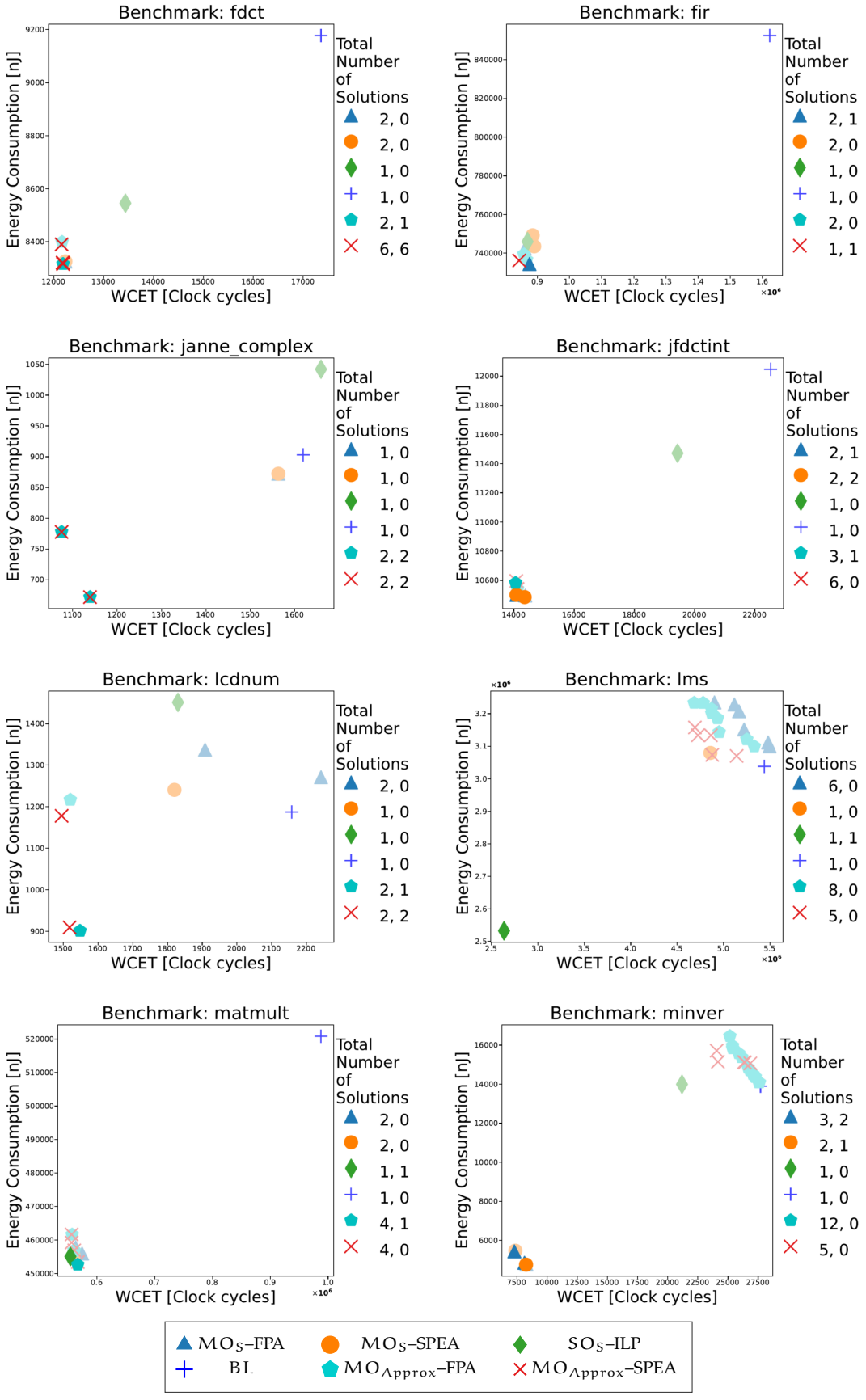
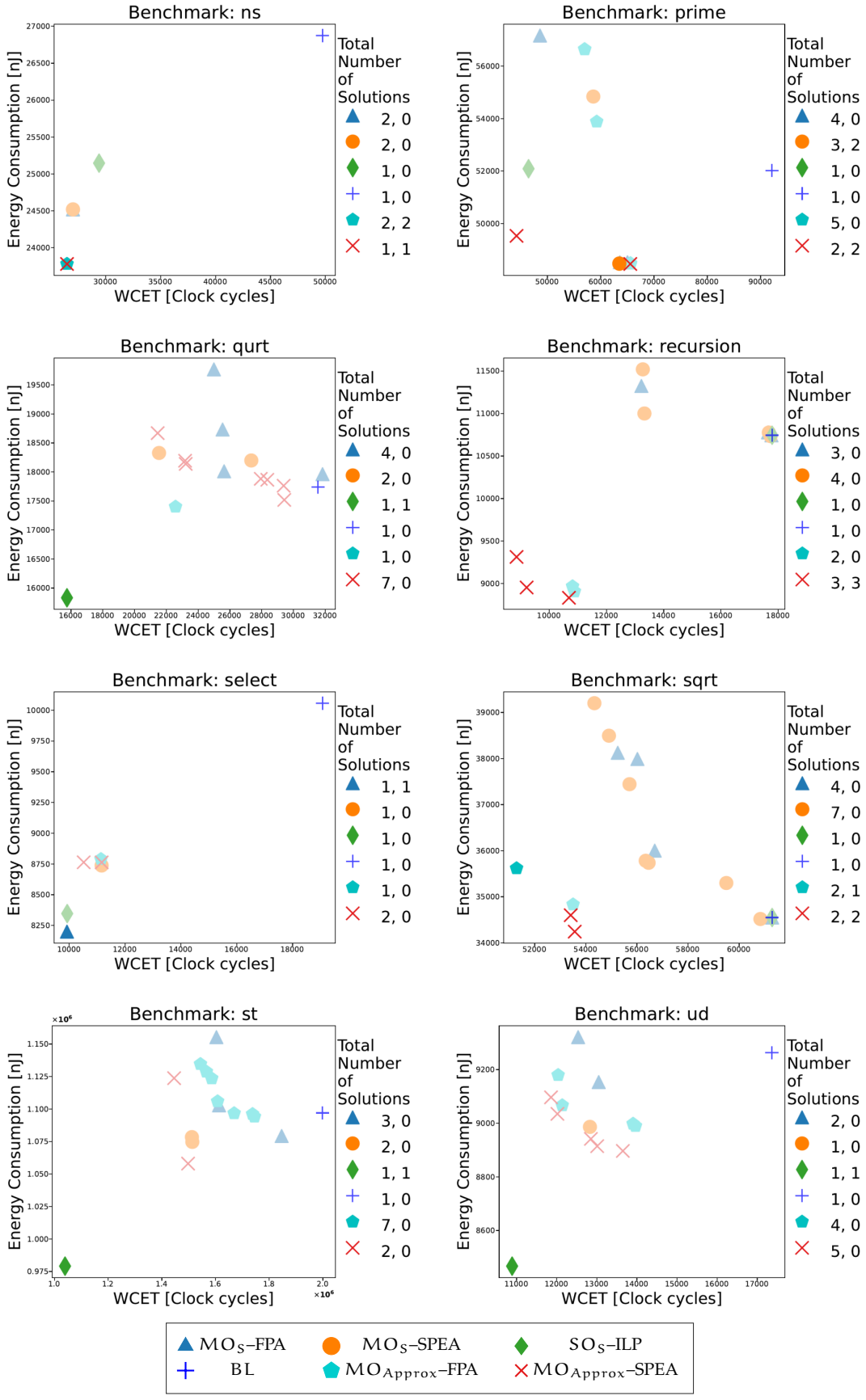


Figure B.3.: (Continued) Pareto fronts for the remaining benchmarks



C

Search Space Reduction of Multi-Objective Static SPM Allocation for Multi-Task Systems

This appendix consists of the following evaluation results of the remaining task sets from the evaluations described in Chapter 7.

- Table C.1 consists of a list of the task sets, the number of tasks in the respective task set, and the names of the single-task MRTC benchmarks that are used to generate multi-task systems for evaluations described in Chapter 7.
- Table C.2 presents the comparison between MTMO, PCA, ICA, and BL approaches using two quality indicators—$\mathcal{C}$ and NR—for all task sets from the evaluations described in Chapter 7.
- Table C.3 presents the comparison between MTMO, PCA, ICA, and BL approaches using RNS and HV for all task sets from the evaluations described in Chapter 7.
- For any task set in Tables C.2 and C.3, the approach with quality indicators in bold performed better (or nondominant) than other approaches.
- Figures C.1, C.2, C.3, C.4, and C.5 present the Pareto fronts for all the remaining task sets from the evaluations described in Chapter 7 for MTMO, PCA, ICA, and BL evaluation runs.

Table C.1.: Characterization of the evaluated task sets

Task Set	#Tasks	Task Names
2TasksMRTC_1	2	matmult, qurt
2TasksMRTC_2	2	expint, fibcall
2TasksMRTC_3	2	expint, st
2TasksMRTC_4	2	fir, st
2TasksMRTC_5	2	compress, insertsort
2TasksMRTC_6	2	adpcm, fac
2TasksMRTC_7	2	expint, fac
2TasksMRTC_8	2	janne_complex, matmult
2TasksMRTC_9	2	bsort100, ndes
2TasksMRTC_10	2	adpcm, expint
4TasksMRTC_1	4	edn, jfdctint, minver, ndes
4TasksMRTC_2	4	edn, matmult, ns, st
4TasksMRTC_3	4	fdct, matmult, qsort-exam, qurt
4TasksMRTC_4	4	cnt, expint, insertsort, sqrt
4TasksMRTC_5	4	insertsort, janne_complex, ns, qurt
4TasksMRTC_6	4	compress, fac, qurt, ud
4TasksMRTC_7	4	compress, edn, ns, st
4TasksMRTC_8	4	edn, fac, insertsort, jfdctint
4TasksMRTC_9	4	expint, janne_complex, minver, ud
4TasksMRTC_10	4	jfdctint, ns, qsort-exam, qurt
6TasksMRTC_1	6	bsort100, cnt, compress, fir, ndes, qsort-exam
6TasksMRTC_2	6	cnt, compress, fibcall, matmult, qsort-exam, sqrt
6TasksMRTC_3	6	adpcm, bsort100, fir, jfdctint, matmult, st
6TasksMRTC_4	6	bsort100, cnt, edn, fac, jfdctint, ns
6TasksMRTC_5	6	fac, fdct, jfdctint, matmult, qurt, ud
6TasksMRTC_6	6	bs, bsort100, cnt, fir, sqrt, st
6TasksMRTC_7	6	compress, expint, fac, fir, jfdctint, matmult
6TasksMRTC_8	6	bsort100, jfdctint, ndes, qurt, sqrt, ud
6TasksMRTC_9	6	bs, cnt, fdct, fir, matmult, minver
6TasksMRTC_10	6	bsort100, cnt, fir, jfdctint, minver, sqrt
8TasksMRTC_1	8	adpcm, bs bsort100, compress, fdct, fibcall, insertsort, st
8TasksMRTC_2	8	bsort100, fac, fdct, fir, jfdctint, matmult, ndes, ud
8TasksMRTC_3	8	bsort100, compress, fibcall, fir, jfdctint, ndes, ns, sqrt
8TasksMRTC_4	8	cnt, edn, fac, fdct, jfdctint, ndes, qsort-exam, st
8TasksMRTC_5	8	edn, fibcall, jfdctint, minver, ndes, ns, qsort-exam, ud
8TasksMRTC_6	8	edn, expint, fac, fdct, insertsort, ns, qurt, ud
8TasksMRTC_7	8	cnt, compress, expint, fdct, fir, jfdctint, qurt, sqrt
8TasksMRTC_8	8	adpcm, bs, compress, edn, expint, qsort-exam, sqrt, st
8TasksMRTC_9	8	bsort100, edn, fir, minver, ns, qurt, st, ud
8TasksMRTC_10	8	adpcm, edn, fdct, minver, ndes, ns, qsort-exam, ud

Table C.2.: Coverage (𝒞) and Non-dominance Ratio (NR) metrics for MTMO, PCA, ICA, and BL evaluations

	Coverage (𝒞)							Non-dominance Ratio (NR)						
	MTMO		PCA		ICA		BL	MTMO		PCA		ICA		BL
	FPA	SPEA	FPA	SPEA	FPA	SPEA		FPA	SPEA	FPA	SPEA	FPA	SPEA	
Benchmarks														
2TasksMRTC_1	1	1	1	1	**0**	1	1	0	0	0	0	**1**	0	0
2TasksMRTC_2	1	0.5	**0**	1	0.5	1	1	0	0.167	**0.667**	0	0.167	0	0
2TasksMRTC_3	1	1	1	1	**0**	1	1	0	0	0	0	**1**	0	0
2TasksMRTC_4	1	1	1	1	**0**	1	1	0	0	0	0	**1**	0	0
2TasksMRTC_5	1	**0**	1	1	1	1	1	0	**1**	0	0	0	0	0
2TasksMRTC_6	0.5	**0**	1	**0**	1	1	1	**0.4**	**0.4**	0	0.2	0	0	0
2TasksMRTC_7	1	1	**0.25**	1	0.667	1	1	0	0	**0.6**	0	0.4	0	0
2TasksMRTC_8	1	1	1	1	**0.875**	1	1	0	0	0	0	**1**	0	0
2TasksMRTC_9	1	**0**	1	1	1	1	1	0	**1**	0	0	0	0	0
2TasksMRTC_10	**0**	0.667	1	1	**0**	1	1	**0.4**	0.2	0	0	**0.4**	0	0
4TasksMRTC_1	**0**	1	1	1	1	1	1	**1**	0	0	0	0	0	0
4TasksMRTC_2	1	1	1	1	**0**	1	1	0	0	0	0	**1**	0	0
4TasksMRTC_3	1	**0**	1	1	1	1	1	0	**1**	0	0	0	0	0
4TasksMRTC_4	1	1	1	1	**0**	1	1	0	0	0	0	**1**	0	0
4TasksMRTC_5	0.5	**0**	1	1	1	0.5	1	**0.4**	**0.4**	0	0	0	0.2	0
4TasksMRTC_6	1	0.667	1	1	**0**	1	1	0	**0.6**	0	0	0.4	0	0
4TasksMRTC_7	1	1	1	1	**0**	1	1	0	0	0	0	**1**	0	0
4TasksMRTC_8	1	1	1	1	**0**	0.5	1	0	0	0	0	**0.833**	0.167	0
4TasksMRTC_9	0.667	1	1	1	**0**	1	1	**0.5**	0	0	0	**0.5**	0	0
4TasksMRTC_10	**0**	1	1	1	**0**	1	1	**0.5**	0	0	0	**0.5**	0	0
6TasksMRTC_1	1	**0**	1	1	1	1	1	0	**1**	0	0	0	0	0
6TasksMRTC_2	1	1	1	1	**0**	1	1	0	0	0	0	**1**	0	0
6TasksMRTC_3	1	1	1	1	**0**	1	1	0	0	0	0	**1**	0	0
6TasksMRTC_4	0.571	1	1	**0**	1	1	1	**0.75**	0	0	0.25	0	0	0
6TasksMRTC_5	**0**	1	1	1	1	1	1	**1**	0	0	0	0	0	0
6TasksMRTC_6	1	0.667	1	1	**0**	1	1	0	0.25	0	0	**0.75**	0	0
6TasksMRTC_7	**0**	1	1	1	1	1	1	**1**	0	0	0	0	0	0
6TasksMRTC_8	0.5	0.5	1	**0**	1	1	1	**0.429**	0.286	0	0.286	0	0	0
6TasksMRTC_9	1	**0**	1	1	1	1	1	0	**1**	0	0	0	0	0
6TasksMRTC_10	**0**	**0**	1	1	1	1	1	0.25	**0.75**	0	0	0	0	0
8TasksMRTC_1	1	**0**	1	1	0.5	1	1	0	**0.667**	0	0	0.333	0	0
8TasksMRTC_2	**0**	1	1	1	1	1	1	**1**	0	0	0	0	0	0
8TasksMRTC_3	**0**	0.6	1	1	1	1	1	**0.5**	**0.5**	0	0	0	0	0
8TasksMRTC_4	**0**	0.5	1	1	0.5	1	1	0.333	0.167	0	0	**0.5**	0	0
8TasksMRTC_5	**0**	0.25	1	1	1	1	1	0.4	**0.6**	0	0	0	0	0
8TasksMRTC_6	0.857	**0**	1	1	1	1	1	0.25	**0.75**	0	0	0	0	0
8TasksMRTC_7	1	1	1	1	**0**	1	1	0	0	0	0	**1**	0	0
8TasksMRTC_8	1	1	1	1	**0**	1	1	0	0	0	0	**1**	0	0
8TasksMRTC_9	1	**0**	0.667	1	1	1	1	0	**0.75**	0.25	0	0	0	0
8TasksMRTC_10	**0.5**	0.75	0.889	1	1	1	1	**0.5**	0.25	0.25	0	0	0	0

Table C.3.: Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for MTMO, PCA, ICA, and BL evaluations

	Ratio of Non-dominated Solutions (RNS)							HyperVolume (HV)						
	MTMO		PCA		ICA		BL	MTMO		PCA		ICA		BL
	FPA	SPEA	FPA	SPEA	FPA	SPEA		FPA	SPEA	FPA	SPEA	FPA	SPEA	
Benchmarks														
2TasksMRTC_1	0	0	0	0	**1**	0	0	0.828	0.965	0.958	0.958	**0.998**	0.954	0
2TasksMRTC_2	0	0.5	**1**	0	0.5	0	0	0.823	0.956	**0.976**	0.952	0.963	0.962	0
2TasksMRTC_3	0	0	0	0	**1**	0	0	0.162	0.246	0.092	0.058	**0.999**	0.561	0
2TasksMRTC_4	0	0	0	0	**1**	0	0	0.299	0.044	0.128	0.164	**0.994**	0.676	0
2TasksMRTC_5	0	**1**	0	0	0	0	0	0.899	**0.988**	0.184	0.494	0.242	0.195	0
2TasksMRTC_6	0.5	**1**	0	**1**	0	0	0	0.601	**0.94**	0.683	0.915	0.637	0.442	0
2TasksMRTC_7	0	0	**0.75**	0	0.333	0	0	0.873	0.866	0.613	0.471	**0.924**	0.867	0
2TasksMRTC_8	0	0	0	0	**0.125**	0	0	0.787	0.757	0.29	0.576	**0.967**	0.45	0
2TasksMRTC_9	0	**1**	0	0	0	0	0	0.751	**0.999**	0.076	0.367	0.573	0.658	0
2TasksMRTC_10	**1**	0.333	0	0	**1**	0	0	**0.968**	**0.968**	0.119	0.427	0.949	0.926	0
4TasksMRTC_1	**1**	0	0	0	0	0	0	**0.994**	0.786	0.073	0.288	0.658	0.472	0
4TasksMRTC_2	0	0	0	0	**1**	0	0	0.325	0.553	0.077	0.076	**0.587**	0.534	0
4TasksMRTC_3	0	**1**	0	0	0	0	0	0.862	**0.998**	0.23	0.519	0.917	0.827	0
4TasksMRTC_4	0	0	0	0	**1**	0	0	0.258	0.571	0.02	0.05	**0.991**	0.686	0
4TasksMRTC_5	0.5	**1**	0	0	0	0.5	0	0.832	0.898	0.367	0.315	0.916	**0.947**	0
4TasksMRTC_6	0	0.333	0	0	**1**	0	0	0.075	0.254	0.119	0.052	**0.336**	0.199	0
4TasksMRTC_7	0	0	0	0	**1**	0	0	0.324	0.402	0.055	0.221	**0.998**	0.294	0
4TasksMRTC_8	0	0	0	0	**1**	0.5	0	0.86	0.861	0.372	0.598	**0.993**	**0.993**	0
4TasksMRTC_9	0.333	0	0	0	**1**	0	0	**0.623**	0.385	0.273	0.222	0.554	0.478	0
4TasksMRTC_10	**1**	0	0	0	**1**	0	0	**0.918**	0.454	0.235	0.221	0.835	0.621	0
6TasksMRTC_1	0	**1**	0	0	0	0	0	0.751	**0.964**	0.28	0.273	0.171	0.292	0
6TasksMRTC_2	0	0	0	0	**1**	0	0	0.26	0.33	0.163	0.32	**0.998**	0.824	0
6TasksMRTC_3	0	0	0	0	**1**	0	0	0.457	0.58	0.052	0.368	**0.999**	0.653	0
6TasksMRTC_4	0.429	0	0	**1**	0	0	0	0.696	0.821	0.007	**0.863**	0.722	0.688	0
6TasksMRTC_5	**1**	0	0	0	0	0	0	**0.998**	0.778	0.245	0.229	0.269	0.328	0
6TasksMRTC_6	0	0.333	0	0	**1**	0	0	0.327	0.595	0.246	0.313	**0.966**	0.764	0
6TasksMRTC_7	**1**	0	0	0	0	0	0	**0.998**	0.858	0.917	0.917	0.89	0.852	0
6TasksMRTC_8	0.5	0.5	0	**1**	0	0	0	0.123	0.483	0.338	**0.929**	0.514	0.56	0
6TasksMRTC_9	0	**1**	0	0	0	0	0	0.817	**0.999**	0.625	0.255	0.388	0.382	0
6TasksMRTC_10	**1**	**1**	0	0	0	0	0	**0.986**	0.959	0.672	0.784	0.313	0.335	0
8TasksMRTC_1	0	**1**	0	0	0.5	0	0	0.161	0.544	0.201	0.238	**0.583**	0.467	0
8TasksMRTC_2	**1**	0	0	0	0	0	0	**0.999**	0.392	0.163	0.098	0.092	0.441	0
8TasksMRTC_3	**1**	0.4	0	0	0	0	0	**0.991**	0.87	0.156	0.145	0.52	0.494	0
8TasksMRTC_4	**1**	0.5	0	0	0.5	0	0	**0.767**	0.679	0.082	0.154	0.262	0.122	0
8TasksMRTC_5	**1**	0.75	0	0	0	0	0	**0.869**	0.412	0.108	0.188	0.129	0.14	0
8TasksMRTC_6	0.143	**1**	0	0	0	0	0	0.781	**0.989**	0.239	0.281	0.687	0.232	0
8TasksMRTC_7	0	0	0	0	**1**	0	0	0.892	0.268	0.27	0.263	**0.999**	0.61	0
8TasksMRTC_8	0	0	0	0	**1**	0	0	0.025	0.046	0.024	0.068	**0.999**	0.613	0
8TasksMRTC_9	0	**1**	0.333	0	0	0	0	0.84	**0.982**	0.154	0.055	0.089	0.061	0
8TasksMRTC_10	**0.5**	0.25	0.111	0	0	0	0	0.533	0.639	0.2	0.22	0.355	**0.988**	0.002

Figure C.1.: Pareto fronts for the remaining task sets using MTMO, PCA, ICA, and BL evaluation runs for multi-task systems

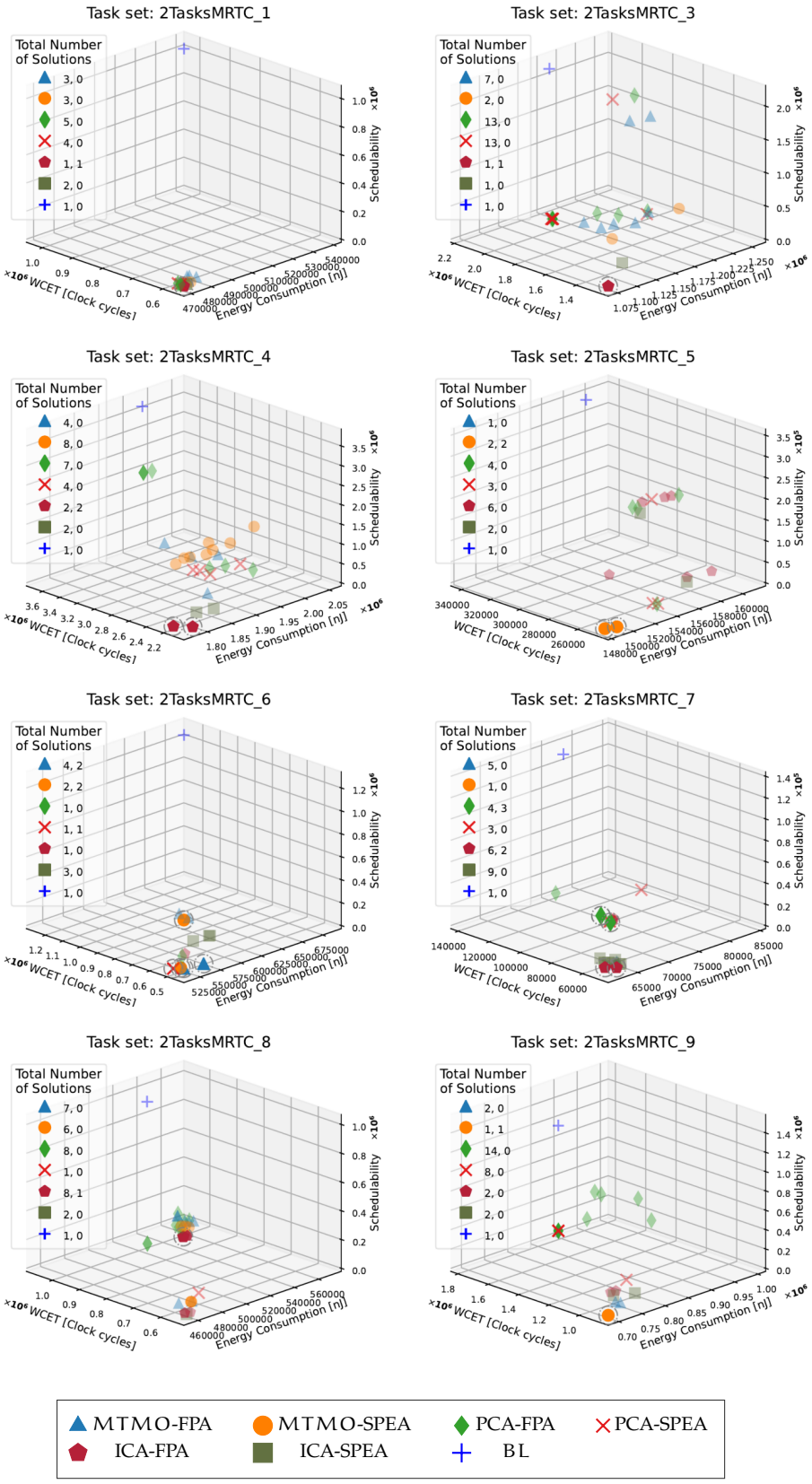


Figure C.2.: (Continued) Pareto fronts for the remaining task sets

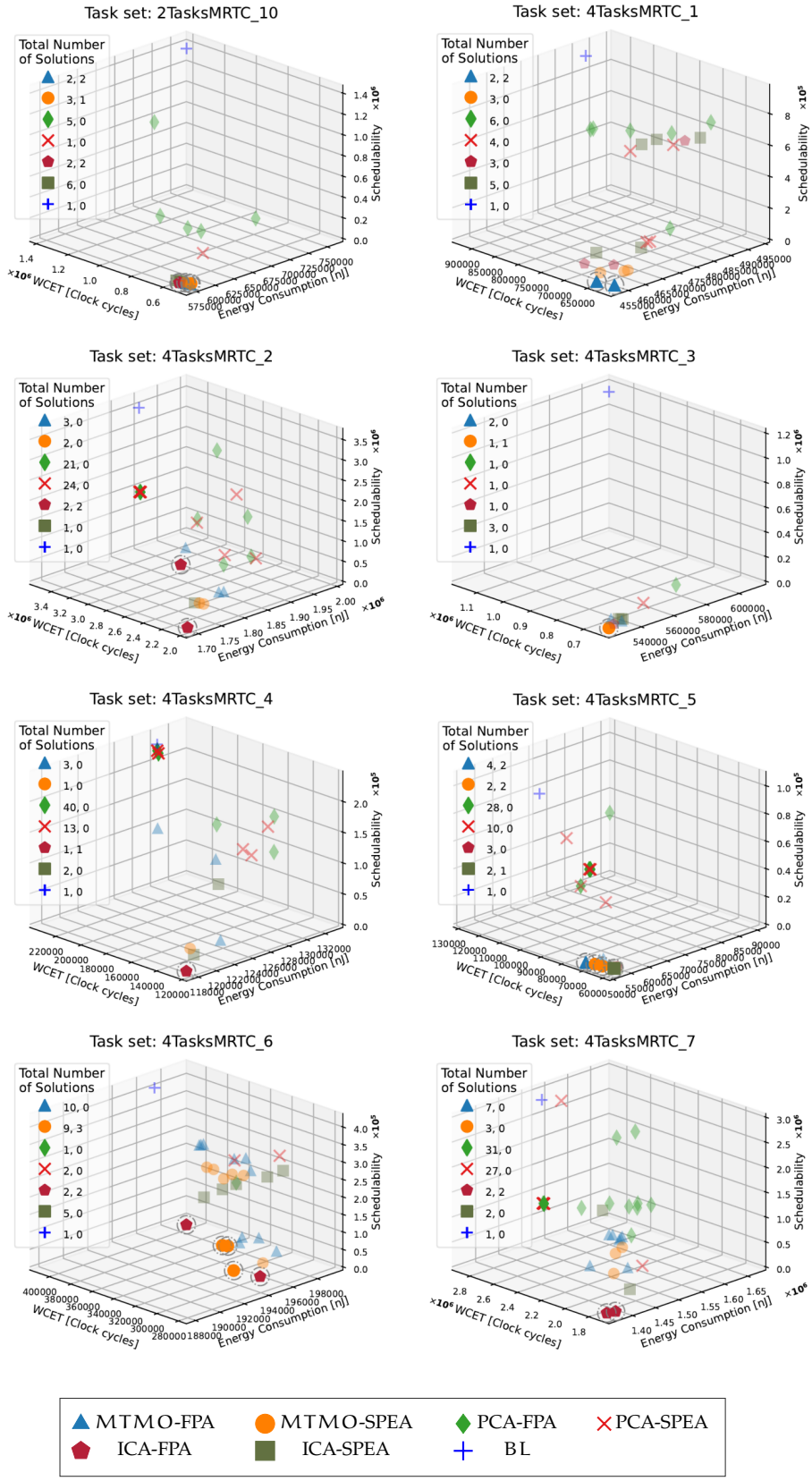


Figure C.3.: (Continued) Pareto fronts for the remaining task sets

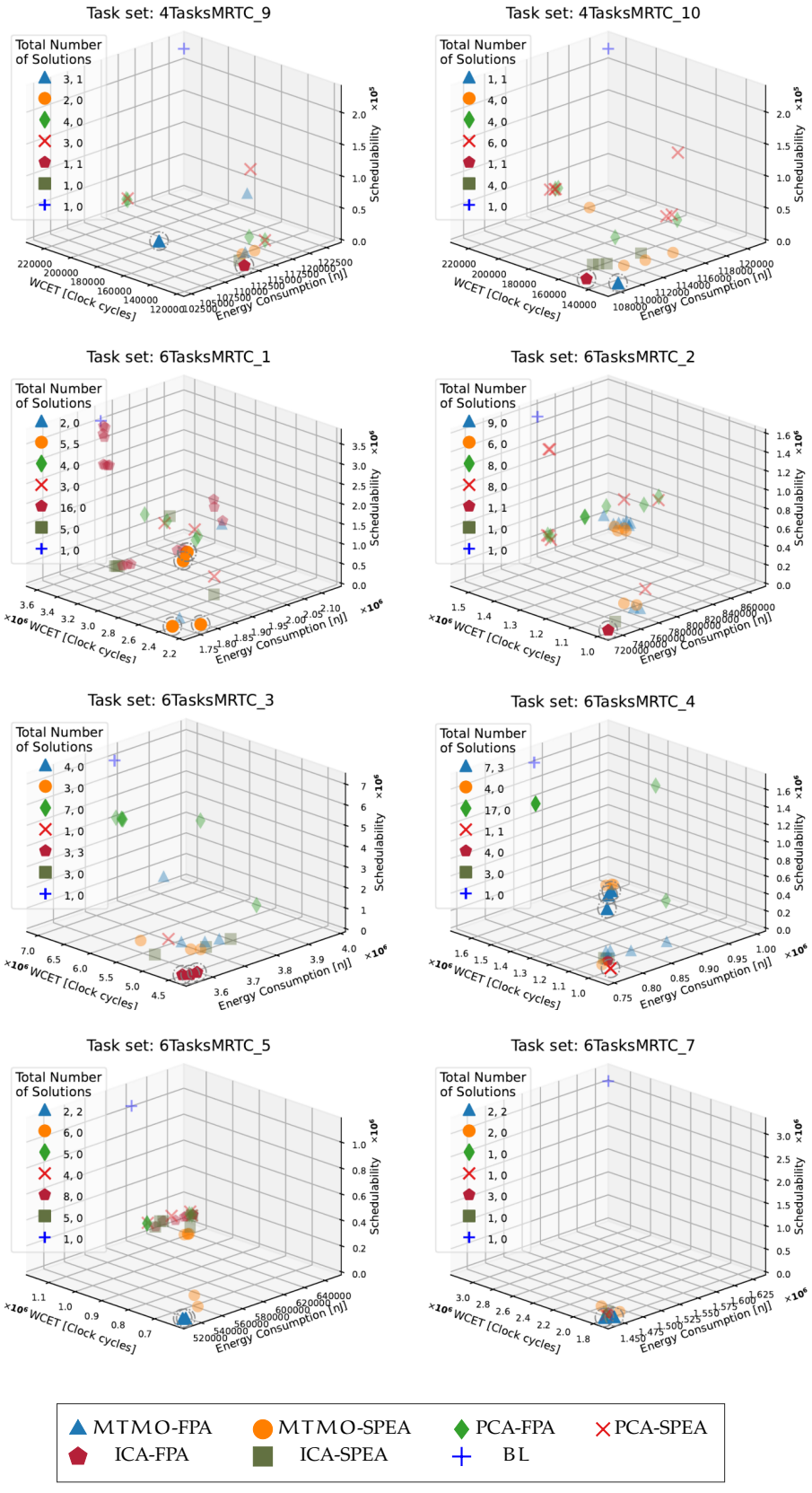


Figure C.4.: (Continued) Pareto fronts for the remaining task sets

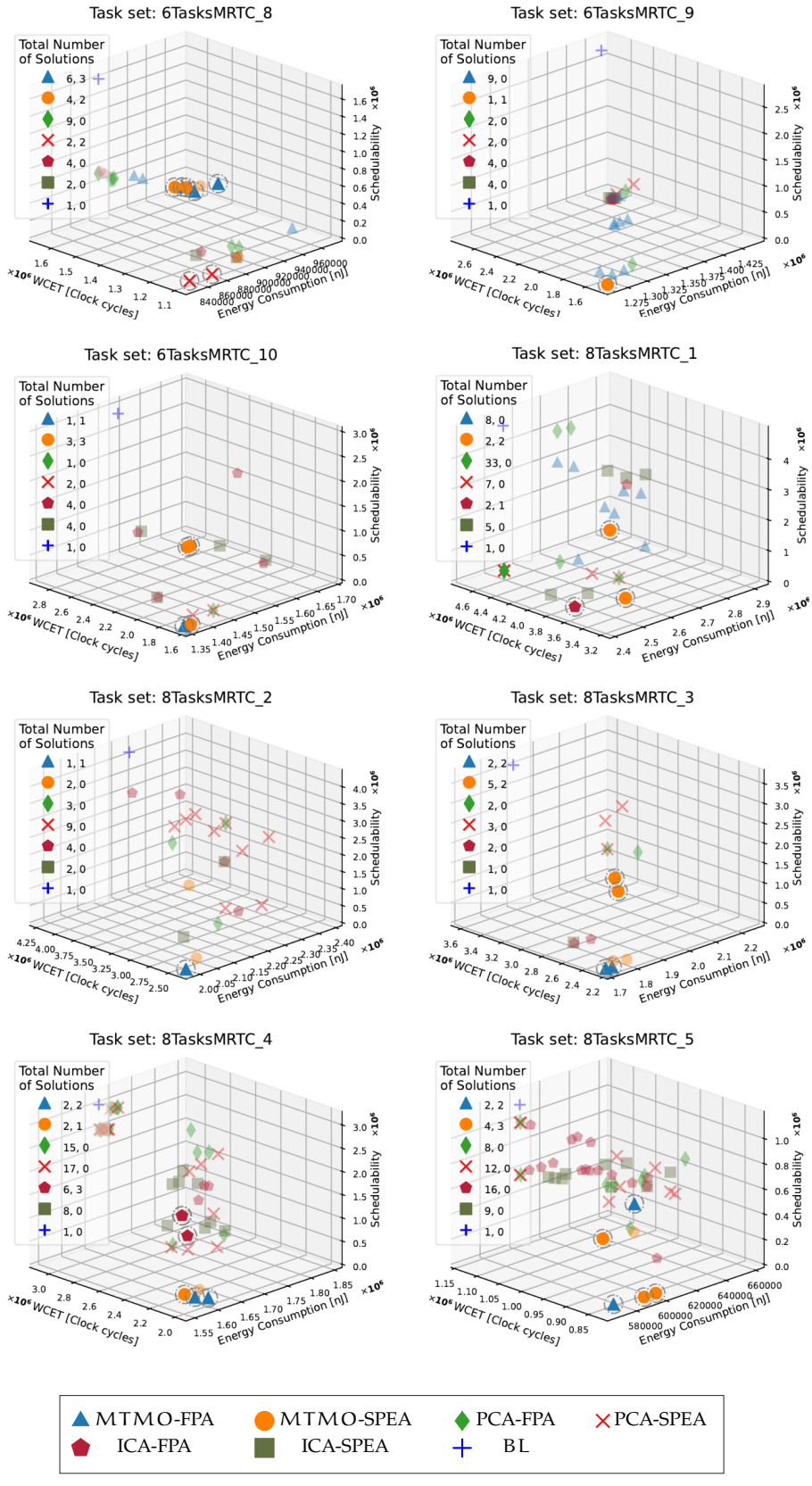
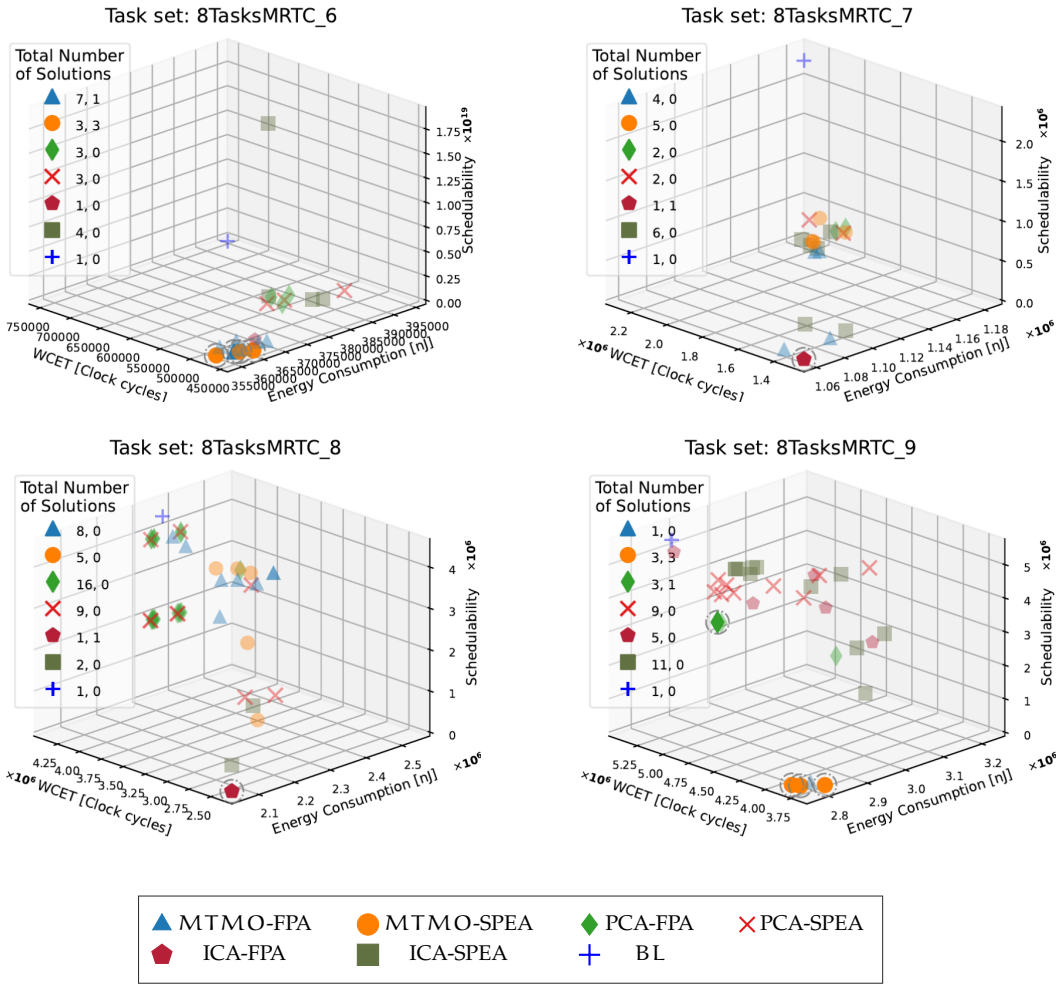


Figure C.5.: (Continued) Pareto fronts for the remaining task sets



D

Multi-Objective Dynamic SPM Allocation

Tables D.1 and D.2 present the quality indicator values for all benchmarks from the evaluations described in Chapter 8 for MO_D, MO_S, SO_D, and BL evaluation runs. For any benchmark, the approach with quality indicators in bold performed better (or nondominant) than other approaches.

Figure D.1 and Figure D.2 present the Pareto fronts for all the remaining benchmarks from the evaluations described in Chapter 8 for MO_D, MO_S, SO_D, and BL evaluation runs.

Table D.1.: Coverage ($\mathcal{C}$) and Non-dominance Ratio (NR) for MO_D, MO_S, SO_D, and BL evaluation runs

	Coverage ($\mathcal{C}$)						Non-dominance Ratio (NR)					
	MO_S		MO_D		SO_D	BL	MO_S		MO_D		SO_D	BL
Benchmarks	FPA	SPEA	FPA	SPEA	ILP		FPA	SPEA	FPA	SPEA	ILP	
Auto_a2time	1	**0**	**0**	**0**	1	**0**	0	0.045	**0.682**	0.227	0	0.045
Auto_aifftr	0.5	**0.2**	1	1	1	1	0.2	**0.8**	0	0	0	0
Auto_aifirf	1	1	**0**	**0**	1	1	0	0	0.25	**0.75**	0	0
Auto_aiifft	1	**0**	1	1	1	1	0	**1**	0	0	0	0
Auto_basefp	1	1	1	1	**0**	1	0	0	0	0	**1**	0
Auto_cacheb	**0**	0.833	1	1	1	1	**0.75**	0.25	0	0	0	0
Auto_canrdr	0.75	**0**	1	1	1	1	0.25	**0.75**	0	0	0	0
Auto_iirflt	1	1	0.706	**0**	1	1	0	0	0.263	**0.737**	0	0
Auto_pntrch	1	1	**0**	**0**	1	1	0	0	0.2	**0.8**	0	0
Auto_puwmod	1	**0**	**0**	**0**	**0**	**0**	0	0.083	**0.625**	0.208	0.042	0.042
Auto_rspeed	1	**0**	1	1	1	1	0	**1**	0	0	0	0
Auto_tblook	1	1	**0**	**0**	1	1	0	0	**0.588**	0.412	0	0
Auto_ttsprk	**0**	0.5	1	1	1	1	**0.667**	0.333	0	0	0	0
Netw_ip_pktcheck	0.5	**0**	1	1	1	1	0.25	**0.75**	0	0	0	0
Netw_ospfv2	1	**0**	1	1	1	1	0	**1**	0	0	0	0
Tele_autocor	**0**	**0**	1	1	1	1	0.2	**0.8**	0	0	0	0
Tele_conven	**0**	**0**	1	1	1	1	**0.5**	**0.5**	0	0	0	0
Tele_fbital	1	1	**0**	**0**	1	1	0	0	**0.5**	**0.5**	0	0
Tele_fft	1	1	**0**	**0**	1	1	0	0	**0.5**	**0.5**	0	0
Tele_viterb	0.5	**0**	1	1	1	1	0.25	**0.75**	0	0	0	0

Table D.2.: Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) for MO_D, MO_S, SO_D, and BL evaluation runs

	Ratio of Non-dominated Solutions (RNS)						HyperVolume (HV)					
	MO_S		MO_D		SO_D	BL	MO_S		MO_D		SO_D	BL
	FPA	SPEA	FPA	SPEA	ILP		FPA	SPEA	FPA	SPEA	ILP	
Benchmarks												
Auto_a2time	0	**1**	**1**	**1**	0	**1**	0.111	**0.642**	0.036	0.036	0.034	0.045
Auto_aifftr	0.5	**0.8**	0	0	0	0	0.998	**1**	0.937	0.973	0	0.002
Auto_aifirf	0	0	**1**	**1**	0	0	0.769	0.901	**1**	**1**	0	0
Auto_aiifft	0	**1**	0	0	0	0	0.923	**1**	0.522	0.534	0.043	0
Auto_basefp	0	0	0	0	**1**	0	0.002	0.033	0.435	0.437	**1**	0
Auto_cacheb	**1**	0.167	0	0	0	0	**1**	0.986	0.755	0	0.749	0.755
Auto_canrdr	0.25	**1**	0	0	0	0	0.847	**0.909**	0.138	0.140	0.043	0
Auto_iirflt	0	0	0.294	**1**	0	0	0.207	0.131	0.574	**0.591**	0.044	0.044
Auto_pntrch	0	0	**1**	**1**	0	0	0.063	0.084	**1**	**1**	0.021	0.021
Auto_puwmod	0	**1**	**1**	**1**	**1**	**1**	0.575	**0.802**	0	0	0	0
Auto_rspeed	0	**1**	0	0	0	0	0.995	**1**	0.97	0.97	0	0.97
Auto_tblook	0	0	**1**	**1**	0	0	0.75	0.779	**1**	**1**	0	0.573
Auto_ttsprk	**1**	0.5	0	0	0	0	**0.969**	0.932	0.013	0.135	0	0
Netw_ip_pktcheck	0.5	**1**	0	0	0	0	0.958	**0.997**	0.001	0.018	0	0
Netw_ospfv2	0	**1**	0	0	0	0	0.719	**1**	0	0	0	0
Tele_autocor	**1**	**1**	0	0	0	0	**1**	**1**	0.742	0.742	0	0.027
Tele_conven	**1**	**1**	0	0	0	0	**0.971**	0.941	0.643	0.643	0	0.267
Tele_fbital	0	0	**1**	**1**	0	0	0.534	0.568	**1**	**1**	0	0
Tele_fft	0	0	**1**	**1**	0	0	0.951	0.945	**1**	**1**	0	0
Tele_viterb	0.5	**1**	0	0	0	0	0.984	**0.994**	0.001	0.001	0	0.001

Figure D.1.: Pareto fronts for the remaining benchmarks using MO_D, MO_S, SO_D, and BL evaluation runs

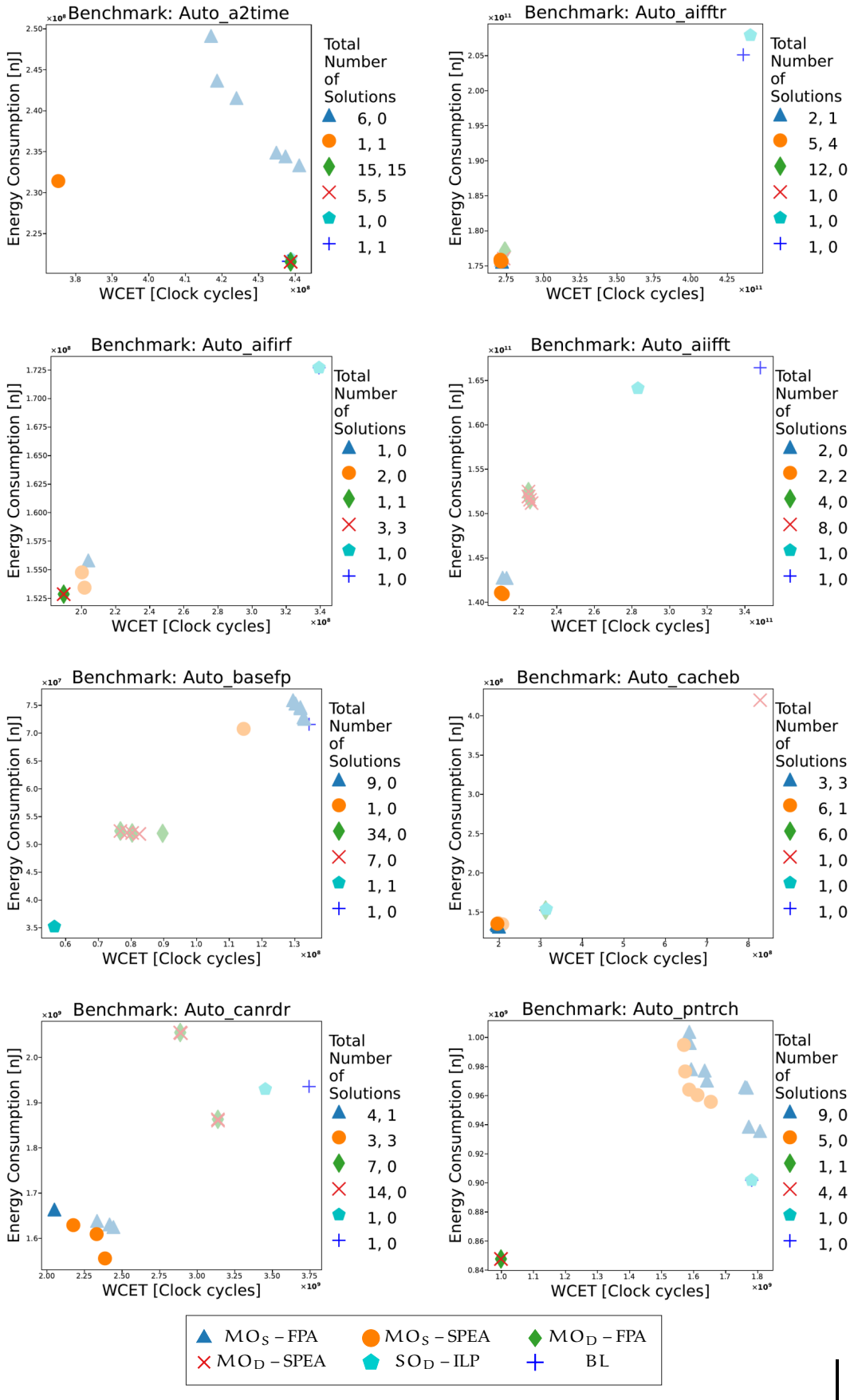
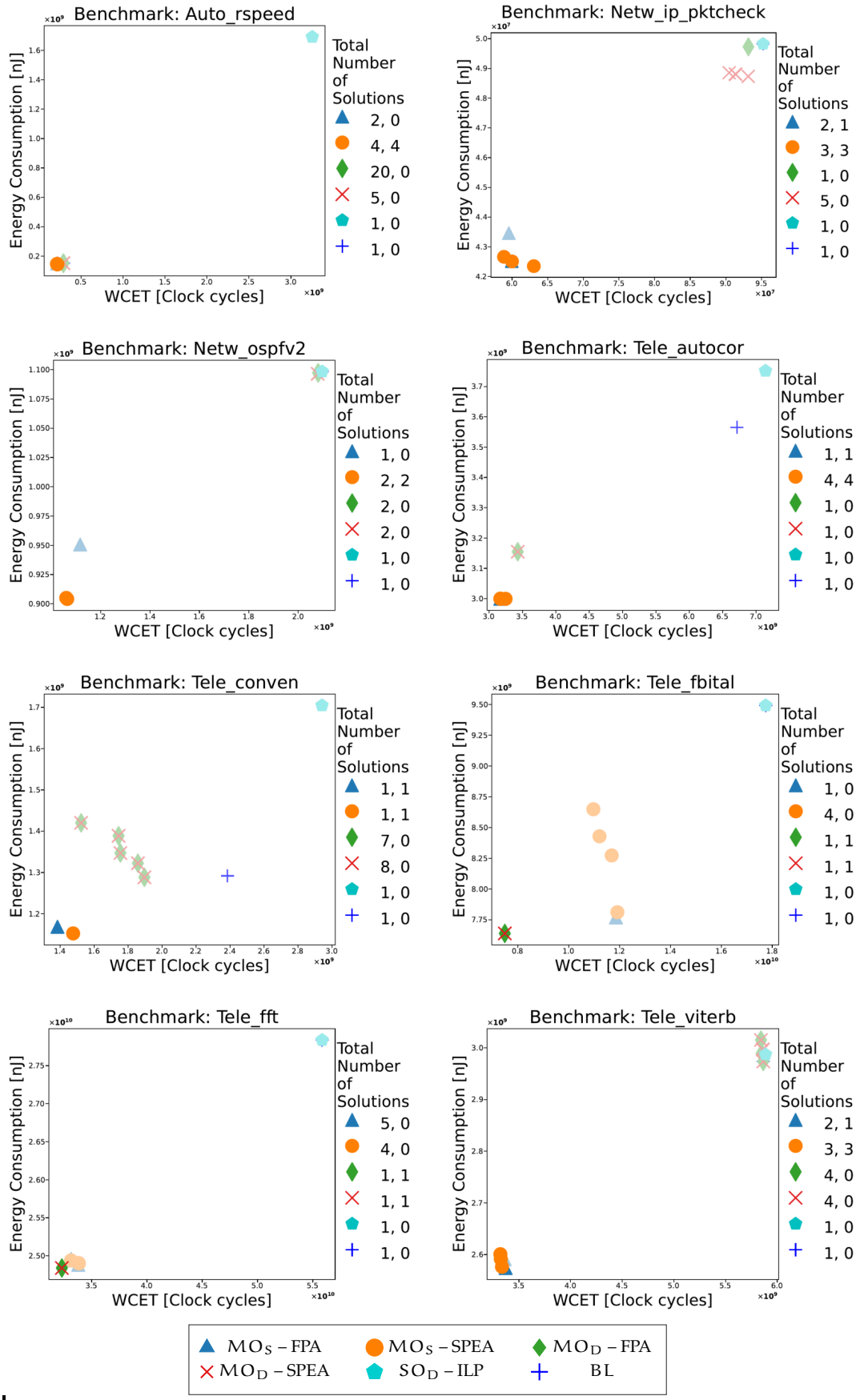


Figure D.2.: (Continued) Pareto fronts for the remaining benchmarks



E

DMA-aware Multi-Objective Dynamic SPM Allocation

This appendix consists of the following evaluation results and figures from the evaluations described in Chapter 9.

- Tables E.1 and E.2 present the quality indicator $\mathcal{C}$, NR, RNS, and HV values, respectively, for all the benchmarks for the 60% relative SPM size and DCPO–FPA configuration evaluations described in Subsection 9.5.3.
- Tables E.3 and E.4 present the quality indicator $\mathcal{C}$, NR, RNS, and HV values, respectively, for the 60% relative SPM size and DCPO–ILP configuration evaluations described in Subsection 9.5.4.
- Tables E.5 and E.6 present the quality indicator values for the 20% relative SPM size and DCPO–FPA configuration; Tables E.7 and E.8 present the quality indicator values for the 40% relative SPM size configuration; Tables E.9 and E.10 present the quality indicator values for the 80% relative SPM size configuration described in Subsection 9.5.5. For any benchmark, the approach with quality indicators in bold performed better (or nondominant) than other approaches.
- Figure E.1 and Figure E.2 present the Pareto fronts for all the remaining benchmarks for the 60% relative SPM size and DCPO–FPA configuration evaluations described in Subsection 9.5.3 for MO_{DMA}, MO_{D}, MO_{S}, SO_{DMA}, SO_{D}, and BL evaluation runs.
- Figure E.3 and Figure E.4 present the Pareto fronts for all the remaining benchmarks for the 60% relative SPM size and DCPO–ILP configuration evaluations described in Subsection 9.5.4.
- Figures E.5, E.6, and E.7 present the Pareto fronts for all the benchmarks for the 20% relative SPM size and DCPO–FPA configuration; Figures E.8, E.9, and E.10 present the Pareto fronts for all the benchmarks for the 40% relative SPM size configuration; Figures E.11, E.12, and E.13 present the Pareto fronts for all the benchmarks for the 80% relative SPM size configuration evaluations described in Subsection 9.5.5.

Table E.1.: Coverage (C) and Non-dominance Ratio (NR) metrics for relative 60% SPM size and DCPO–FPA

	Coverage (C)									Non-dominance Ratio (NR)								
	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA		FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
Benchmarks																		
Auto_a2time	1	**0**	1	1	1	1	**0**	**0**	1	0	0.023	0	0	0	0	**0.75**	0.227	0
Auto_aifftr	0.5	**0.2**	1	1	1	1	1	1	1	0.2	**0.8**	0	0	0	0	0	0	0
Auto_aifirf	1	1	1	1	1	1	**0**	0.5	1	0	0	0	0	0	0	**0.958**	0.042	0
Auto_aiifft	1	**0**	1	1	1	1	1	1	1	0	**1**	0	0	0	0	0	0	0
Auto_basefp	1	1	1	1	**0**	1	**0**	1	1	0	0	0	0	0.2	0	**0.8**	0	0
Auto_cacheb	1	1	1	1	1	1	1	**0.167**	1	0	0	0	0	0	0	0	**1**	0
Auto_canrdr	0.75	**0**	1	1	1	1	1	0.25	1	0.143	**0.429**	0	0	0	0	0	**0.429**	0
Auto_iirflt	1	1	1	1	1	1	0.4375	**0**	1	0	0	0	0	0	0	0.375	**0.625**	0
Auto_pntrch	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	0.357	**0.643**	0
Auto_puwmod	1	**0**	1	1	1	**0**	0.735	0.444	1	0	0.118	0	0	0	0.058	**0.529**	0.294	0
Auto_rspeed	1	**0**	1	1	1	**0**	**0**	**0**	1	0	0.093	0	0	0	0.023	**0.767**	0.116	0
Auto_tblook	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.922**	0.078	0
Auto_ttsprk	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.5**	**0.5**	0
Netw_ip_pktcheck	0.5	**0**	1	1	1	1	**0**	1	1	0.2	**0.6**	0	0	0	0	0.2	0	0
Netw_ospfv2	1	**0**	1	1	1	1	**0**	**0**	1	0	**0.5**	0	0	0	0	0.25	0.25	0
Tele_autocor	**0**	**0**	1	1	1	1	1	1	1	0.2	**0.8**	0	0	0	0	0	0	0
Tele_conven	**0**	**0**	1	1	1	1	1	1	1	**0.5**	**0.5**	0	0	0	0	0	0	0
Tele_fbital	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.5**	**0.5**	0
Tele_fft	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.5**	**0.5**	0
Tele_viterb	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	0.333	**0.667**	0

Table E.2.: Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for relative 60% SPM size and DCPO–FPA

	Ratio of Non-dominated Solutions (RNS)									HyperVolume (HV)								
	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA		FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
Benchmarks																		
Auto_a2time	0	**1**	0	0	0	0	**1**	**1**	0	0.093	0.535	0.03	0.03	0.028	0.655	**0.67**	**0.67**	0.037
Auto_aifftr	0.5	**0.8**	0	0	0	0	0	0	0	0.998	**1**	0.937	0.973	0	0.002	0	0	0.002
Auto_aifirf	0	0	0	0	0	0	**1**	0.5	0	0.545	0.639	0.709	0.709	0	0.001	**1**	**1**	0
Auto_aiifft	0	**1**	0	0	0	0	0	0	0	0.923	**1**	0.522	0.534	0.043	0	0.535	0.548	0
Auto_basefp	0	0	0	0	**1**	0	**1**	0	0	0.002	0.032	0.429	0.431	**0.987**	0.003	0.654	0.654	0
Auto_cacheb	0	0	0	0	0	0	0	**0.833**	0	0.972	0.959	0.734	0	0.728	0.956	0.998	**1**	0.734
Auto_canrdr	0.25	**1**	0	0	0	0	0	0.75	0	0.847	**0.909**	0.138	0.14	0.043	0.286	0.75	0.756	0
Auto_iirflt	0	0	0	0	0	0	0.5625	**1**	0	0.153	0.097	0.423	0.436	0.033	0.03	0.754	**0.77**	0
Auto_pntrch	0	0	0	0	0	0	**1**	**1**	0	0.063	0.083	0.986	0.986	0.021	0.021	**1**	**1**	0.021
Auto_puwmod	0	**1**	0	0	0	**1**	0.265	0.556	0	0.225	0.314	0	0	0	0.626	**0.753**	**0.753**	0.004
Auto_rspeed	0	**1**	0	0	0	**1**	**1**	**1**	0	0.994	**0.999**	0.969	0.969	0	0.996	0.996	0.996	0.969
Auto_tblook	0	0	0	0	0	0	**1**	**1**	0	0.74	0.768	0.986	0.986	0	0.565	**1**	**1**	0.565
Auto_ttsprk	0	0	0	0	0	0	**1**	**1**	0	0.606	0.583	0.008	0.084	0	0	**1**	**1**	0
Netw_ip_pktcheck	0.5	**1**	0	0	0	0	**1**	0	0	0.765	**0.796**	0.001	0.015	0	0	0.757	0.749	0
Netw_ospfv2	0	**1**	0	0	0	0	**1**	**1**	0	0.767	**0.997**	0.004	0.004	0	0	0.889	0.889	0
Tele_autocor	**1**	**1**	0	0	0	0	0	0	0	**1**	**1**	0.742	0.742	0	0.027	0.775	0.775	0.027
Tele_conven	**1**	**1**	0	0	0	0	0	0	0	**0.971**	0.941	0.643	0.643	0	0	0.708	0.708	0.267
Tele_fbital	0	0	0	0	0	0	**1**	**1**	0	0.524	0.557	0.982	0.982	0	0.831	**1**	**1**	0
Tele_fft	0	0	0	0	0	0	**1**	**1**	0	0.936	0.931	0.985	0.985	0	0	**1**	**1**	0
Tele_viterb	0	0	0	0	0	0	**1**	**1**	0	0.93	0.94	0.001	0.001	0	0.028	0.999	**1**	0.001

Table E.3.: Coverage (C) and Non-dominance Ratio (NR) metrics for relative 60% SPM size and DCPO–ILP

	Coverage (C)									Non-dominance Ratio (NR)								
	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA		FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
Benchmarks																		
Auto_a2time	1	**0**	1	1	1	1	**0**	**0**	1	0	0.024	0	0	0	0	**0.762**	0.214	0
Auto_aifftr	0.5	0.2	1	1	1	1	1	**0**	1	0.167	**0.667**	0	0	0	0	0	0.167	0
Auto_aifirf	1	1	1	1	1	1	**0.833**	1	1	0	0	0	0	0	0	**1**	0	0
Auto_aiifft	1	**0**	1	1	1	1	1	1	1	0	**1**	0	0	0	0	0	0	0
Auto_basefp	1	1	1	1	**0**	1	**0**	**0**	1	0	0	0	0	0.027	0	**0.919**	0.054	0
Auto_cacheb	1	1	1	1	1	1	1	**0**	1	0	0	0	0	0	0	0	**1**	0
Auto_canrdr	0.75	**0**	1	1	1	1	1	0.25	1	0.143	**0.429**	0	0	0	0	0	**0.429**	0
Auto_iirflt	1	1	1	1	1	1	0.75	**0**	1	0	0	0	0	0	0	0.25	**0.75**	0
Auto_pntrch	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.643**	0.357	0
Auto_puwmod	1	**0**	1	1	1	1	**0**	**0**	1	0	0.043	0	0	0	0	**0.761**	0.196	0
Auto_rspeed	1	**0**	1	1	1	**0**	**0**	**0**	1	0	0.089	0	0	0	0.022	**0.778**	0.111	0
Auto_tblook	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.923**	0.077	0
Auto_ttsprk	1	1	1	1	1	1	**0**	1	1	0	0	0	0	0	0	**1**	0	0
Netw_ip_pktcheck	0.5	**0**	1	1	1	1	0.5	0.5	1	0.167	**0.5**	0	0	0	0	0.167	0.167	0
Netw_ospfv2	1	**0**	1	1	1	1	**0**	**0**	1	0	**0.5**	0	0	0	0	0.25	0.25	0
Tele_autocor	**0**	**0**	1	1	1	1	1	1	1	0.2	**0.8**	0	0	0	0	0	0	0
Tele_conven	**0**	**0**	1	1	1	1	1	1	1	**0.5**	**0.5**	0	0	0	0	0	0	0
Tele_fbital	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.5**	**0.5**	0
Tele_fft	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	0.333	**0.667**	0
Tele_viterb	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.5**	**0.5**	0

Table E.4.: Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for relative 60% SPM size and DCPO–ILP

	Ratio of Non-dominated Solutions (RNS)									HyperVolume (HV)								
	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA		FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
Benchmarks																		
Auto_a2time	0	**1**	0	0	0	0	**1**	**1**	0	0.093	0.535	0.03	0.03	0.028	0.655	**0.67**	**0.67**	0.037
Auto_aifftr	0.50	0.8	0	0	0	0	0	**1**	0	0.994	**0.996**	0.934	0.969	0	0.002	0.985	0.986	0.002
Auto_aifirf	0	0	0	0	0	0	**0.167**	0	0	0.545	0.639	0.709	0.709	0	0.001	**1**	**1**	0
Auto_aiifft	0	**1**	0	0	0	0	0	0	0	0.923	**1**	0.522	0.534	0.043	0	0.535	0.548	0
Auto_basefp	0	0	0	0	**1**	0	**1**	**1**	0	0.002	0.032	0.429	0.431	**0.987**	0.003	0.654	0.654	0
Auto_cacheb	0	0	0	0	0	0	0	**1**	0	0.973	0.959	0.734	0	0.728	0.956	0.998	**1**	0.734
Auto_canrdr	0.25	**1**	0	0	0	0	0	0.75	0	0.847	**0.909**	0.138	0.140	0.043	0.286	0.753	0.756	0
Auto_iirflt	0	0	0	0	0	0	0.25	**1**	0	0.153	0.097	0.425	0.437	0.033	0.03	0.719	**0.772**	0.033
Auto_pntrch	0	0	0	0	0	0	**1**	**1**	0	0.063	0.083	0.986	0.986	0.021	0.021	**1**	**1**	0.021
Auto_puwmod	0	**1**	0	0	0	0	**1**	**1**	0	0.224	0.313	0	0	0	0.623	**0.756**	**0.756**	0
Auto_rspeed	0	**1**	0	0	0	**1**	**1**	**1**	0	0.994	**0.999**	0.969	0.969	0	0.996	0.996	0.996	0.969
Auto_tblook	0	0	0	0	0	0	**1**	**1**	0	0.74	0.768	0.986	0.986	0	0.565	**1**	**1**	0.565
Auto_ttsprk	0	0	0	0	0	0	**1**	0	0	0.596	0.573	0.008	0.083	0	0	**1**	0.995	0
Netw_ip_pktcheck	0.5	**1**	0	0	0	0	0.5	0.5	0	0.765	**0.797**	0.001	0.015	0	0	0.742	0.742	0
Netw_ospfv2	0	**1**	0	0	0	0	**1**	**1**	0	0.767	**0.997**	0.004	0.004	0	0	0.889	0.889	0
Tele_autocor	**1**	**1**	0	0	0	0	0	0	0	**1**	**1**	0.742	0.742	0	0.027	0.838	0.838	0.027
Tele_conven	**1**	**1**	0	0	0	0	0	0	0	**0.971**	0.941	0.643	0.643	0	0	0.737	0.756	0.267
Tele_fbital	0	0	0	0	0	0	**1**	**1**	0	0.524	0.557	0.982	0.982	0	0.831	**1**	**1**	0
Tele_fft	0	0	0	0	0	0	**1**	**1**	0	0.936	0.931	0.985	0.985	0	0	**1**	**1**	0
Tele_viterb	0	0	0	0	0	0	**1**	**1**	0	0.917	0.927	0.001	0.001	0	0.028	**1**	**1**	0.001

Table E.5.: Coverage (C) and Non-dominance Ratio (NR) metrics for relative 20% SPM size and DCPO–FPA

	Coverage (C)									Non-dominance Ratio (NR)								
	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA		FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
Benchmarks																		
Auto_a2time	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.667**	0.333	0
Auto_aifftr	1	1	1	1	1	1	1	**0**	1	0	0	0	0	0	0	0	**1**	0
Auto_aifirf	1	1	1	1	1	1	**0.469**	0.778	1	0	0	0	0	0	0	**0.895**	0.105	0
Auto_aiifft	1	**0**	1	1	1	1	**0**	1	1	0	0.1	0	0	0	0	**0.9**	0	0
Auto_basefp	1	1	1	1	1	1	0.024	**0**	1	0	0	0	0	0	0	**0.872**	0.128	0
Auto_cacheb	1	1	1	1	1	1	0.5	**0**	1	0	0	0	0	0	0	**0.667**	0.333	0
Auto_canrdr	**0**	1	1	1	1	1	**0**	**0**	1	0.071	0	0	0	0	0	0.286	**0.643**	0
Auto_iirflt	1	1	1	1	1	1	0.615	**0**	1	0	0	0	0	0	0	0.4	**0.6**	0
Auto_pntrch	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.857**	0.143	0
Auto_puwmod	1	**0**	1	1	1	1	1	1	1	0	**1**	0	0	0	0	0	0	0
Auto_rspeed	1	1	1	1	1	**0**	1	1	1	0	0	0	0	0	**1**	0	0	0
Auto_tblook	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.894**	0.106	0
Auto_ttsprk	1	**0**	1	1	1	1	1	1	1	0	**1**	0	0	0	0	0	0	0
Netw_ip_pktcheck	1	1	1	1	1	1	1	**0**	1	0	0	0	0	0	0	0	**1**	0
Netw_ospfv2	1	1	1	1	1	1	0.4	**0**	1	0	0	0	0	0	0	**0.6**	0.4	0
Tele_autocor	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.5**	**0.5**	0
Tele_conven	1	1	0.75	0.8	1	1	**0.25**	**0.25**	1	0	0	0.125	0.125	0	0	**0.375**	**0.375**	0
Tele_fbital	**0**	**0**	1	1	1	1	1	1	1	0.333	**0.667**	0	0	0	0	0	0	0
Tele_fft	1	1	1	1	1	1	0.5	**0**	1	0	0	0	0	0	0	0.333	**0.667**	0
Tele_viterb	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	0.333	**0.667**	0

Table E.6.: Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for relative 20% SPM size and DCPO–FPA

	Ratio of Non-dominated Solutions (RNS)									HyperVolume (HV)								
	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA		FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
Benchmarks																		
Auto_a2time	0	0	0	0	0	0	**1**	**1**	0	0.164	0.152	0.002	0.002	0	0.96	**0.997**	**0.997**	0.011
Auto_aifftr	0	0	0	0	0	0	0	**1**	0	0.478	0.478	0.962	0.962	0	0	0.963	**1**	0.003
Auto_aifirf	0	0	0	0	0	0	**0.531**	0.222	0	0.249	0.248	0.730	0	0	0	**1**	**1**	0
Auto_aiifft	0	**1**	0	0	0	0	**1**	0	0	0.652	0.801	0.927	0.927	0.054	0	**0.941**	**0.941**	0.065
Auto_basefp	0	0	0	0	0	0	0.976	**1**	0	0.005	0.006	0.484	0.484	0.574	0	0.984	**0.995**	0
Auto_cacheb	0	0	0	0	0	0	0.5	**1**	0	0.931	0.953	0.782	0	0	0.928	**0.999**	**0.999**	0.782
Auto_canrdr	**1**	0	0	0	0	0	**1**	**1**	0	0.748	0.512	0.164	0.076	0.05	0.336	**0.889**	**0.889**	0
Auto_iirflt	0	0	0	0	0	0	0.384615	**1**	0	0.069	0.09	0.453	0.47	0	0.23	0.676	**0.711**	0
Auto_pntrch	0	0	0	0	0	0	**1**	**1**	0	0.199	0.191	0.989	0.989	0.170	0	**1**	**1**	0.17
Auto_puwmod	0	**1**	0	0	0	0	0	0	0	0.621	**1**	0	0	0	0.459	0.421	0.421	0
Auto_rspeed	0	0	0	0	0	**1**	0	0	0	0.978	0.978	0.972	0.972	0	**1**	0.999	0.999	0.972
Auto_tblook	0	0	0	0	0	0	**1**	**1**	0	0.635	0.632	0.986	0.986	0	0.558	**1**	**1**	0.565
Auto_ttsprk	0	**1**	0	0	0	0	0	0	0	0.519	**1**	0	0	0	0	0	0	0
Netw_ip_pktcheck	0	0	0	0	0	0	0	**1**	0	0.628	0.634	0.001	0.013	0	0	0.879	**1**	0
Netw_ospfv2	0	0	0	0	0	0	0.6	**1**	0	0.060	0.276	0.005	0.005	0	0.970	**1**	**1**	0
Tele_autocor	0	0	0	0	0	0	**1**	**1**	0	0.034	0.034	0.995	0.995	0	0	**1**	**1**	0.034
Tele_conven	0	0	0.25	0.2	0	0	**0.75**	**0.75**	0	0.402	0.402	0.898	0.898	0	0.081	0.912	**0.913**	0.315
Tele_fbital	**1**	**1**	0	0	0	0	0	0	0	**1**	**1**	0.721	0.721	0.721	0	0.721	0.721	0.721
Tele_fft	0	0	0	0	0	0	0.5	**1**	0	0.946	0.946	0.935	0.936	0.932	0	0.997	**0.999**	0.932
Tele_viterb	0	0	0	0	0	0	**1**	**1**	0	0.641	0.468	0.162	0.162	0	0.207	**1**	**1**	0.157

Table E.7.: Coverage (C) and Non-dominance Ratio (NR) metrics for relative 40% SPM size and DCPO–FPA

	Coverage (C)									Non-dominance Ratio (NR)								
	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA		FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
Benchmarks																		
Auto_a2time	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.808**	0.192	0
Auto_aifftr	1	**0**	1	**0**	1	1	1	1	1	0	**0.667**	0	0.333	0	0	0	0	0
Auto_aifirf	1	1	1	1	1	1	**0.407**	0.8	1	0	0	0	0	0	0	**0.889**	0.111	0
Auto_aiifft	1	**0**	1	1	1	1	1	1	1	0	**1**	0	0	0	0	0	0	0
Auto_basefp	1	1	1	1	**0**	1	0.054	0.571	1	0	0	0	0	0.026	0	**0.897**	0.077	0
Auto_cacheb	1	1	1	1	1	1	1	**0.333**	1	0	0	0	0	0	0	0	**1**	0
Auto_canrdr	1	0.5	1	1	1	1	0.7	**0**	1	0	0.167	0	0	0	0	0.25	**0.583**	0
Auto_iirflt	1	1	1	1	1	1	0.28	**0**	1	0	0	0	0	0	0	**0.581**	0.419	0
Auto_pntrch	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.737**	0.263	0
Auto_puwmod	**0**	**0**	1	1	1	1	0.703	0.571	1	0.063	0.063	0	0	0	0	**0.688**	0.188	0
Auto_rspeed	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.792**	0.208	0
Auto_tblook	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.959**	0.041	0
Auto_ttsprk	1	1	1	1	1	1	**0**	1	1	0	0	0	0	0	0	**1**	0	0
Netw_ip_pktcheck	0.667	1	1	1	1	1	**0**	1	1	0.143	0	0	0	0	0	**0.857**	0	0
Netw_ospfv2	1	1	1	1	1	1	**0**	0.333	1	0	0	0	0	0	0	0.333	**0.667**	0
Tele_autocor	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.5**	**0.5**	0
Tele_conven	1	**0**	1	1	1	1	0.75	0.667	1	0	**0.333**	0	0	0	0	**0.333**	**0.333**	0
Tele_fbital	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.5**	**0.5**	0
Tele_fft	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.5**	**0.5**	0
Tele_viterb	1	1	1	1	1	1	0.5	**0**	1	0	0	0	0	0	0	0.2	**0.8**	0

Table E.8.: Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for relative 40% SPM size and DCPO–FPA

	Ratio of Non-dominated Solutions (RNS)									HyperVolume (HV)								
	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA		FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
Benchmarks																		
Auto_a2time	0	0	0	0	0	0	**1**	**1**	0	0.029	0.329	0.002	0.002	0	0.968	**0.998**	**0.998**	0.012
Auto_aifftr	0	**1**	0	**1**	0	0	0	0	0	0.977	0.997	0.963	**0.999**	0	0	0	0	0.003
Auto_aifirf	0	0	0	0	0	0	**0.593**	0.2	0	0.438	0.434	0.734	0.735	0	0	**1**	**1**	0
Auto_aiifft	0	**1**	0	0	0	0	0	0	0	0.922	**1**	0.689	0.689	0.206	0	0.69	0.689	0.044
Auto_basefp	0	0	0	0	**1**	0	0.946	0.429	0	0.001	0.002	0.662	0.662	**0.941**	0.001	0.832	0.829	0
Auto_cacheb	0	0	0	0	0	0	0	**0.667**	0	0.937	0.949	0.766	0	0.729	0.911	0.998	**1**	0.735
Auto_canrdr	0	0.5	0	0	0	0	0.3	**1**	0	0.608	**0.927**	0.17	0.17	0.052	0.37	0.917	0.917	0
Auto_iirflt	0	0	0	0	0	0	0.72	**1**	0	0.144	0.09	0.4	0.416	0	0.009	0.755	**0.761**	0
Auto_pntrch	0	0	0	0	0	0	**1**	**1**	0	0.068	0.076	0.986	0.986	0.078	0	**1**	**1**	0.078
Auto_puwmod	**1**	**1**	0	0	0	0	0.297	0.429	0	0.712	0.565	0	0	0	0.555	**0.767**	**0.767**	0
Auto_rspeed	0	0	0	0	0	0	**1**	**1**	0	0.991	0.997	0.972	0.972	0	0.999	**1**	**1**	0.972
Auto_tblook	0	0	0	0	0	0	**1**	**1**	0	0.705	0.699	0.986	0.986	0	0.596	**1**	**1**	0.565
Auto_ttsprk	0	0	0	0	0	0	**1**	0	0	0.596	0.243	0	0.093	0	0	**1**	0.991	0
Netw_ip_pktcheck	0.333	0	0	0	0	0	**1**	0	0	0.642	0.623	0.004	0.012	0	0	**0.991**	0.979	0
Netw_ospfv2	0	0	0	0	0	0	**1**	0.667	0	0.415	0.426	0.008	0.008	0	0	**1**	**1**	0
Tele_autocor	0	0	0	0	0	0	**1**	**1**	0	0.034	0.034	0.958	0.958	0	0	**0.999**	**0.999**	0.033
Tele_conven	0	**1**	0	0	0	0	0.25	0.333	0	0.674	**0.925**	0.709	0.709	0	0	0.762	0.765	0.29
Tele_fbital	0	0	0	0	0	0	**1**	**1**	0	0.226	0.138	0.988	0.988	0	0	**1**	**1**	0
Tele_fft	0	0	0	0	0	0	**1**	**1**	0	0.993	0.993	0.996	0.996	0.005	0	**1**	**1**	0.005
Tele_viterb	0	0	0	0	0	0	0.5	**1**	0	0.381	0.828	0.019	0.651	0	0.04	0.996	**1**	0.002

Table E.9.: Coverage (C) and Non-dominance Ratio (NR) metrics for relative 80% SPM size and DCPO–FPA

	Coverage (C)									Non-dominance Ratio (NR)								
	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA		FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
Benchmarks																		
Auto_a2time	1	**0**	1	1	1	1	**0**	**0**	1	0	0.039	0	0	0	0	**0.784**	0.176	0
Auto_aifftr	1	1	1	**0**	1	1	1	1	1	0	0	0	**1**	0	0	0	0	0
Auto_aifirf	1	1	1	1	1	1	**0**	0.25	1	0	0	0	0	0	0	**0.897**	0.103	0
Auto_aiifft	0.5	**0**	1	1	1	1	1	1	1	0.333	**0.667**	0	0	0	0	0	0	0
Auto_basefp	1	1	1	1	**0**	1	**0**	1	1	0	0	0	0	0.045	0	**0.955**	0	0
Auto_cacheb	**0**	1	1	1	1	1	0.667	**0**	1	0.25	0	0	0	0	0	0.25	**0.5**	0
Auto_canrdr	0.5	1	1	1	1	1	0.667	**0.333**	1	0.167	0	0	0	0	0	**0.5**	0.333	0
Auto_iirflt	1	1	1	1	1	1	0.75	**0.077**	1	0	0	0	0	0	0	0.333	**0.667**	0
Auto_pntrch	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.524**	0.476	0
Auto_puwmod	**0**	0.5	1	1	1	**0**	1	1	1	**0.4**	**0.4**	0	0	0	0.2	0	0	0
Auto_rspeed	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.895**	0.105	0
Auto_tblook	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.919**	0.081	0
Auto_ttsprk	**0**	1	1	1	1	1	**0**	**0**	1	**0.429**	0	0	0	0	0	**0.429**	0.143	0
Netw_ip_pktcheck	**0**	0.667	1	1	1	1	**0**	1	1	0.25	0.25	0	0	0	0	**0.5**	0	0
Netw_ospfv2	**0**	0.5	1	1	1	1	1	1	1	**0.5**	**0.5**	0	0	0	0	0	0	0
Tele_autocor	0.833	**0**	1	1	1	1	**0**	**0**	1	0.2	**0.4**	0	0	0	0	0.2	0.2	0
Tele_conven	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.5**	**0.5**	0
Tele_fbital	1	1	1	1	1	1	**0**	**0**	1	0	0	0	0	0	0	**0.5**	**0.5**	0
Tele_fft	1	**0**	1	1	1	1	1	1	1	0	**1**	0	0	0	0	0	0	0
Tele_viterb	1	1	1	1	1	1	**0**	0.5	1	0	0	0	0	0	0	**0.667**	0.333	0

Table E.10.: Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for relative 80% SPM size and DCPO–FPA

	Ratio of Non-dominated Solutions (RNS)									HyperVolume (HV)								
	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL	MO_S		MO_D		SO_D	SO_{DMA}	MO_{DMA}		BL
	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA		FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
Benchmarks																		
Auto_a2time	0	**1**	0	0	0	0	**1**	**1**	0	0.004	0.325	0.001	0.001	0	0.425	**0.439**	**0.439**	0.005
Auto_aifftr	0	0	0	**1**	0	0	0	0	0	0.002	0.002	0.963	**1**	0	0	0	0	0.002
Auto_aifirf	0	0	0	0	0	0	**1**	0.75	0	0.826	0.844	0.708	0	0	0.002	**1**	**1**	0
Auto_aiifft	0.5	**1**	0	0	0	0	0	0	0	0.997	**1**	0.659	0.659	0.214	0	0.66	0.668	0.052
Auto_basefp	0	0	0	0	**1**	0	**1**	0	0	0.003	0.056	0.5	0.5	0.9	0.004	0.759	0.725	0
Auto_cacheb	**1**	0	0	0	0	0	0.333	**1**	0	**0.972**	0.85	0.001	0	0	0.79	0.95	0.963	0.001
Auto_canrdr	0.5	0	0	0	0	0	0.333	**0.667**	0	**0.939**	0.801	0.177	0	0.043	0.279	0.915	0.915	0
Auto_iirflt	0	0	0	0	0	0	0.25	**0.923**	0	0.364	0.357	0.402	0.415	0	0.154	0.757	**0.76**	0
Auto_pntrch	0	0	0	0	0	0	**1**	**1**	0	0.443	0.213	0.985	0.985	0.09	0	0.999	**1**	0.09
Auto_puwmod	**1**	0.5	0	0	0	**1**	0	0	0	0.261	0.391	0	0	0	**0.685**	0.631	0.631	0
Auto_rspeed	0	0	0	0	0	0	**1**	**1**	0	0.553	0.287	0	0	0	0.209	**1**	**1**	0
Auto_tblook	0	0	0	0	0	0	**1**	**1**	0	0	0	0.972	0.972	0	0.561	**1**	**1**	0.531
Auto_ttsprk	**1**	0	0	0	0	0	**1**	**1**	0	**1**	0.885	0.011	0.011	0	0	0.994	0.994	0
Netw_ip_pktcheck	**1**	0.333	0	0	0	0	**1**	0	0	0.832	0.847	0.005	0.011	0	0	**0.979**	0.96	0
Netw_ospfv2	**1**	0.5	0	0	0	0	0	0	0	**1**	0.99	0	0	0	0.831	0.828	0.828	0
Tele_autocor	0.167	**1**	0	0	0	0	**1**	**1**	0	0.998	**1**	0.82	0.82	0	0.833	0.999	0.999	0.026
Tele_conven	0	0	0	0	0	0	**1**	**1**	0	0.866	0.922	0.812	0.812	0.166	0	**1**	**1**	0.166
Tele_fbital	0	0	0	0	0	0	**1**	**1**	0	0.911	0.989	0.982	0.982	0	0	**1**	**1**	0
Tele_fft	0	**1**	0	0	0	0	0	0	0	**1**	**1**	0.988	0.988	0.993	0	0.894	0.894	0.018
Tele_viterb	0	0	0	0	0	0	**1**	0.5	0	0.983	0.941	0.001	0.001	0	0.025	**1**	**1**	0.001

Figure E.1.: Pareto fronts for the remaining benchmarks using MO_D, MO_S, SO_S, SO_{DMA}, BL, and MO_{DMA} optimization approaches for the relative 60% SPM size and DCPO–FPA case

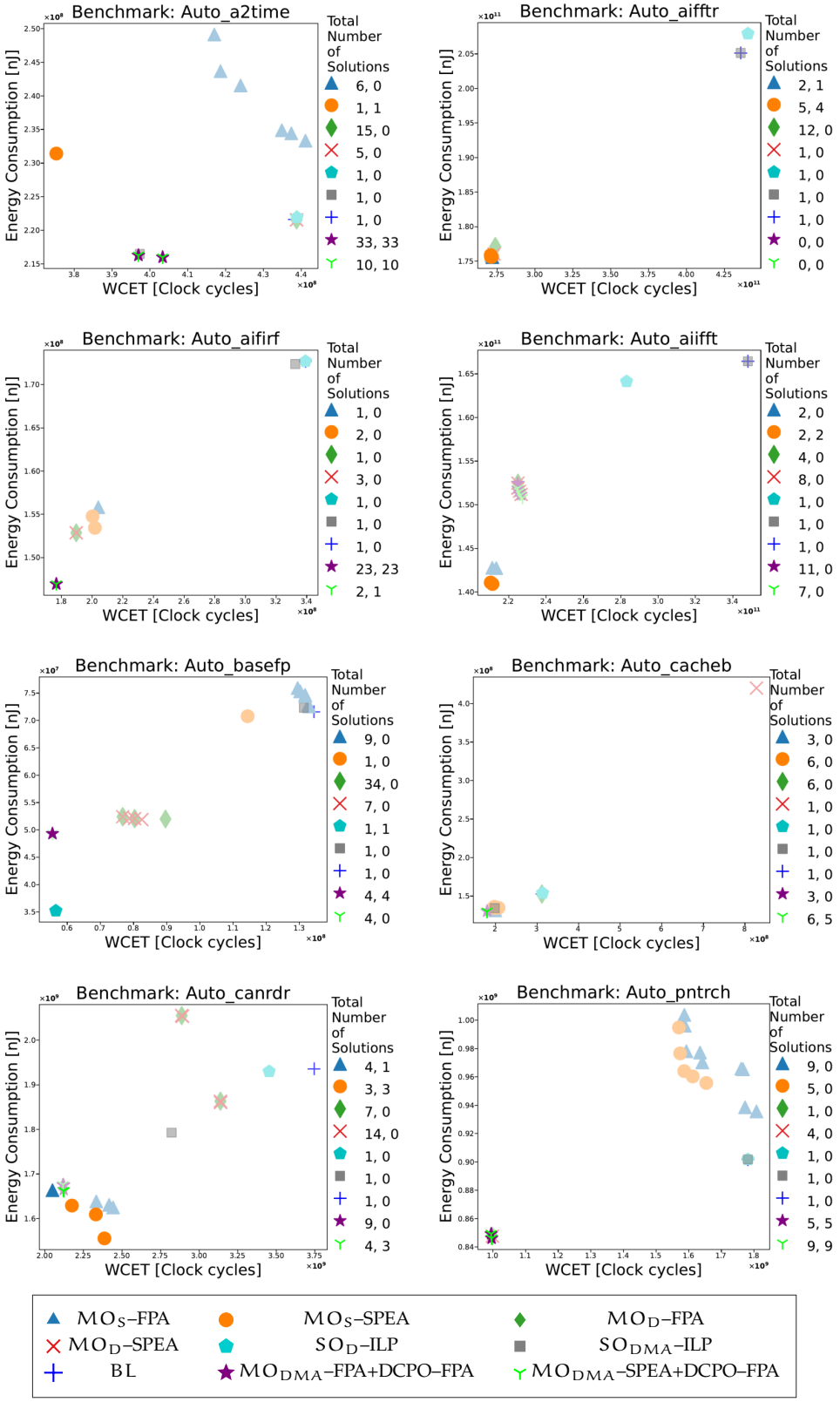


Figure E.2.: (Continued) Pareto fronts for the remaining benchmarks for the relative 60% SPM size and DCPO–FPA case

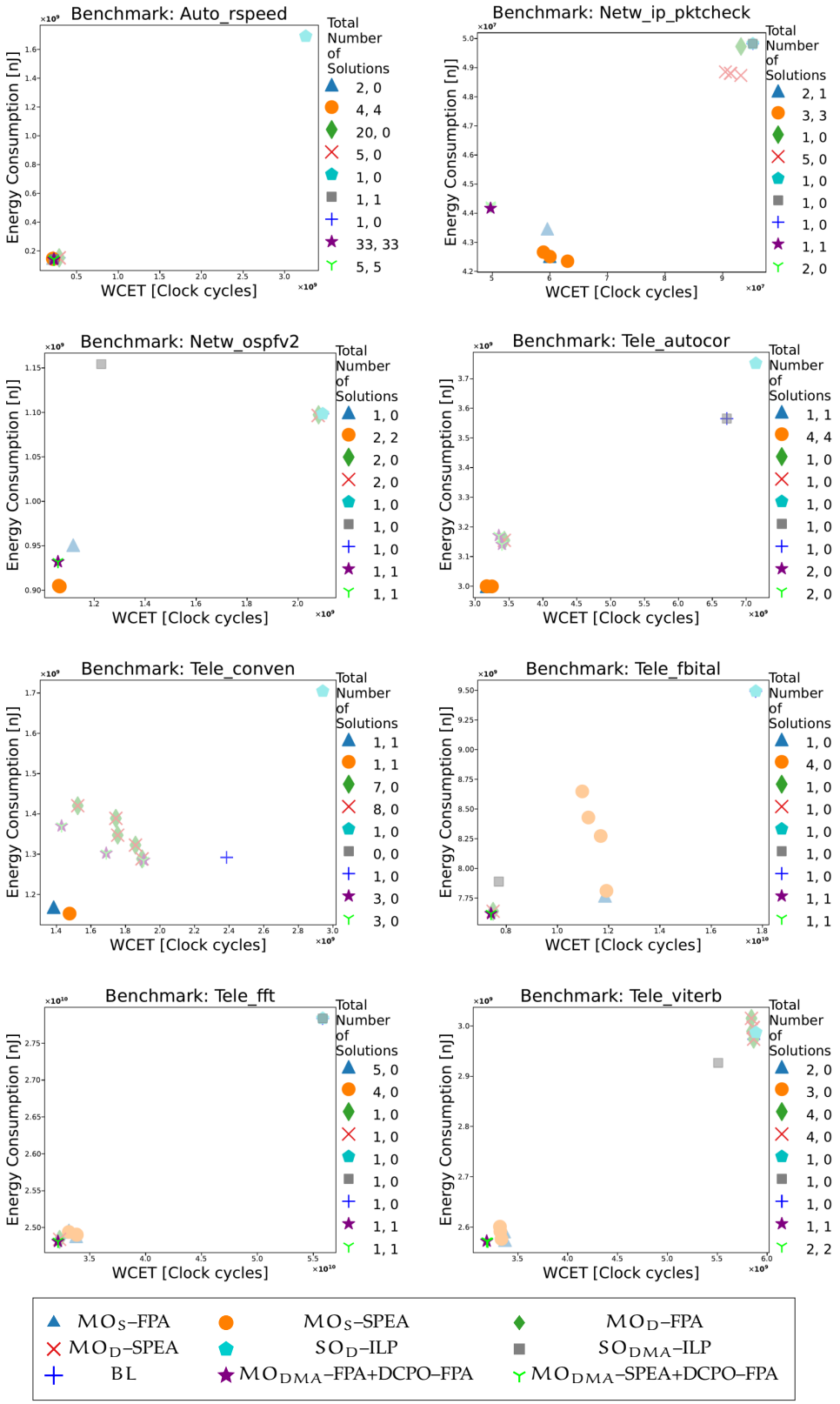


Figure E.3.: Pareto fronts for the remaining benchmarks using MO_D, MO_S, SO_S, SO_{DMA}, BL, and MO_{DMA} optimization approaches for the relative 60% SPM size and DCPO–ILP case

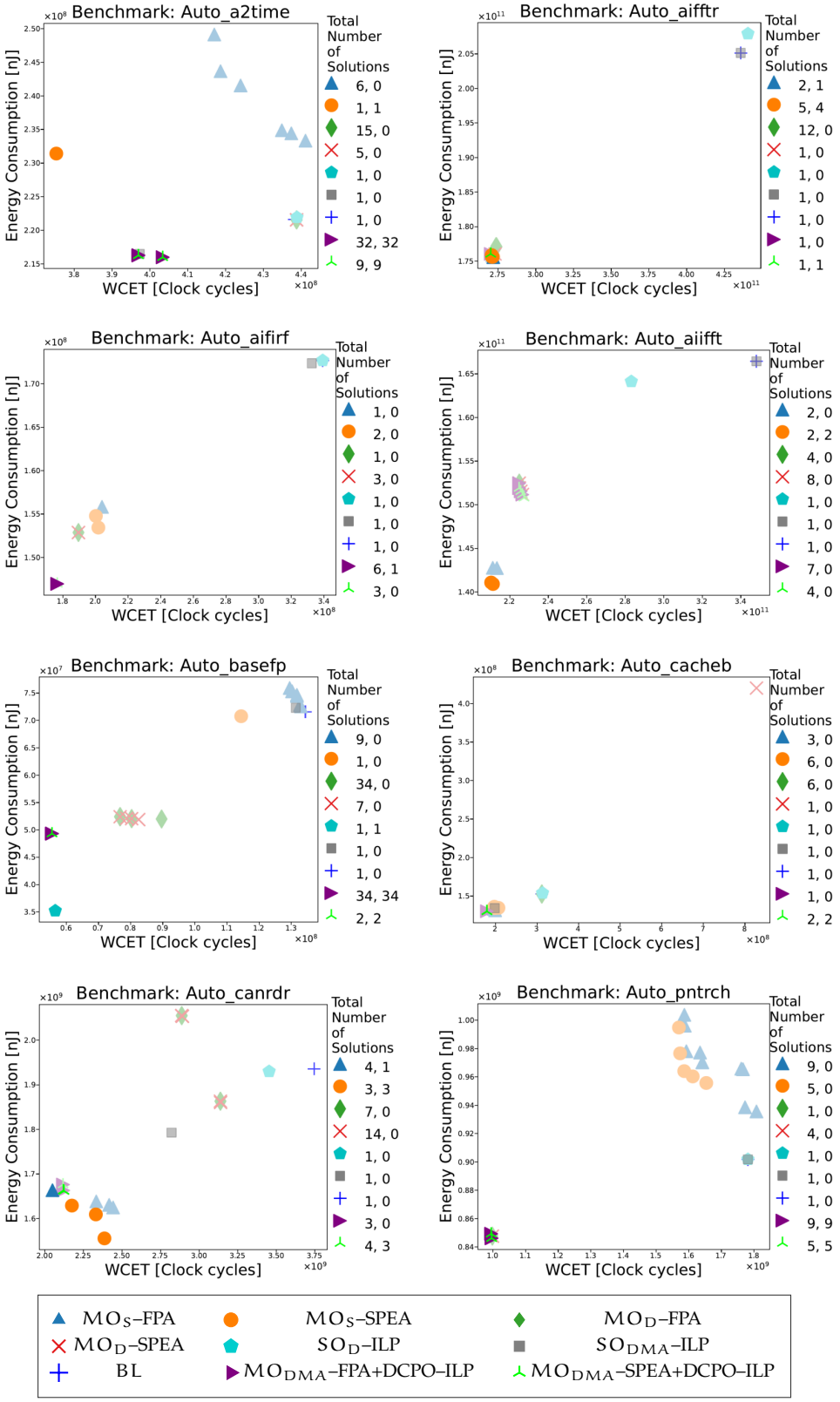


Figure E.4.: (Continued) Pareto fronts for the remaining benchmarks for the relative 60% SPM size and DCPO–ILP case

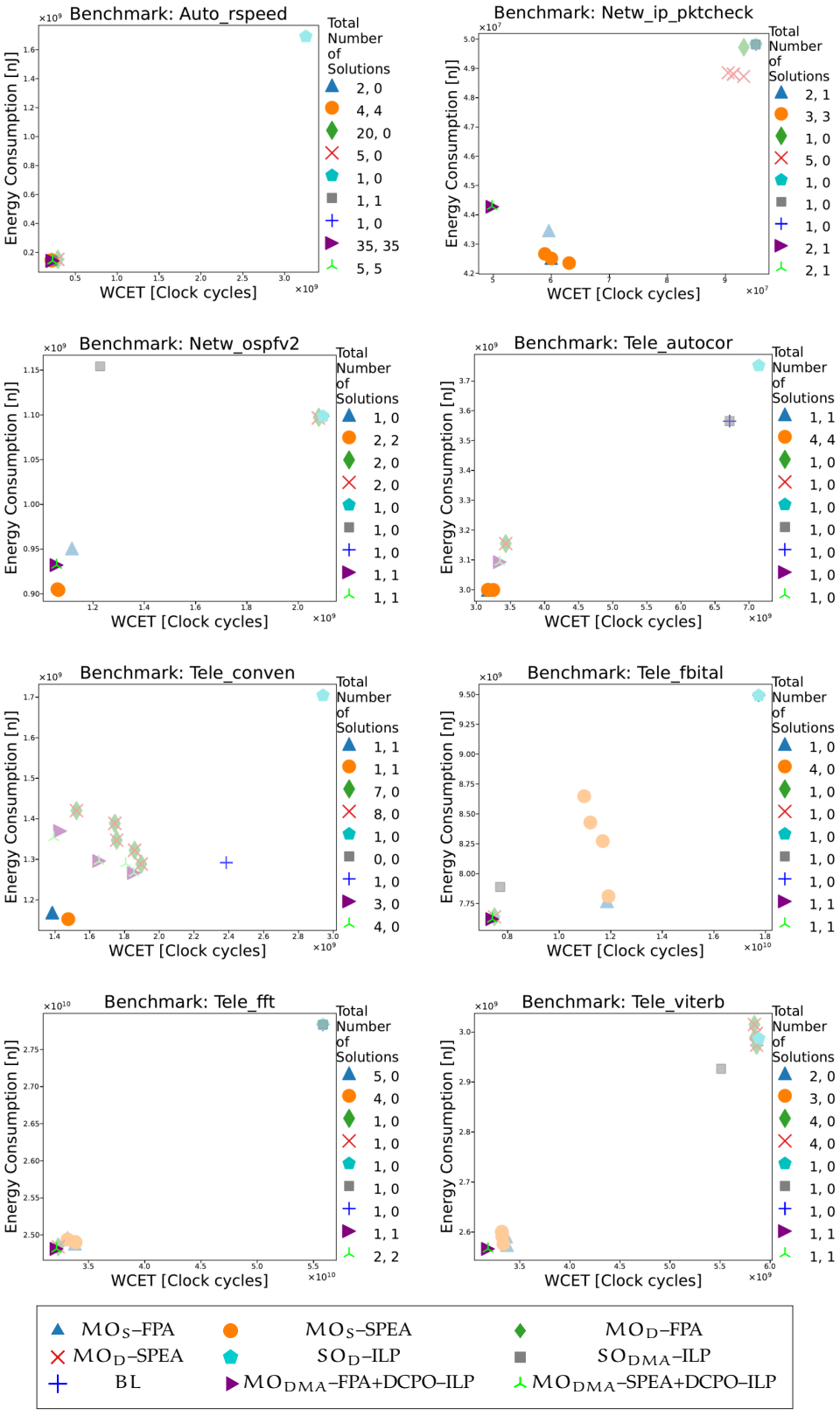


Figure E.5.: Pareto fronts for all the benchmarks using MO_D, MO_S, SO_S, SO_{DMA}, BL, and MO_{DMA} optimization approaches for the relative 20% SPM size and DCPO–FPA case

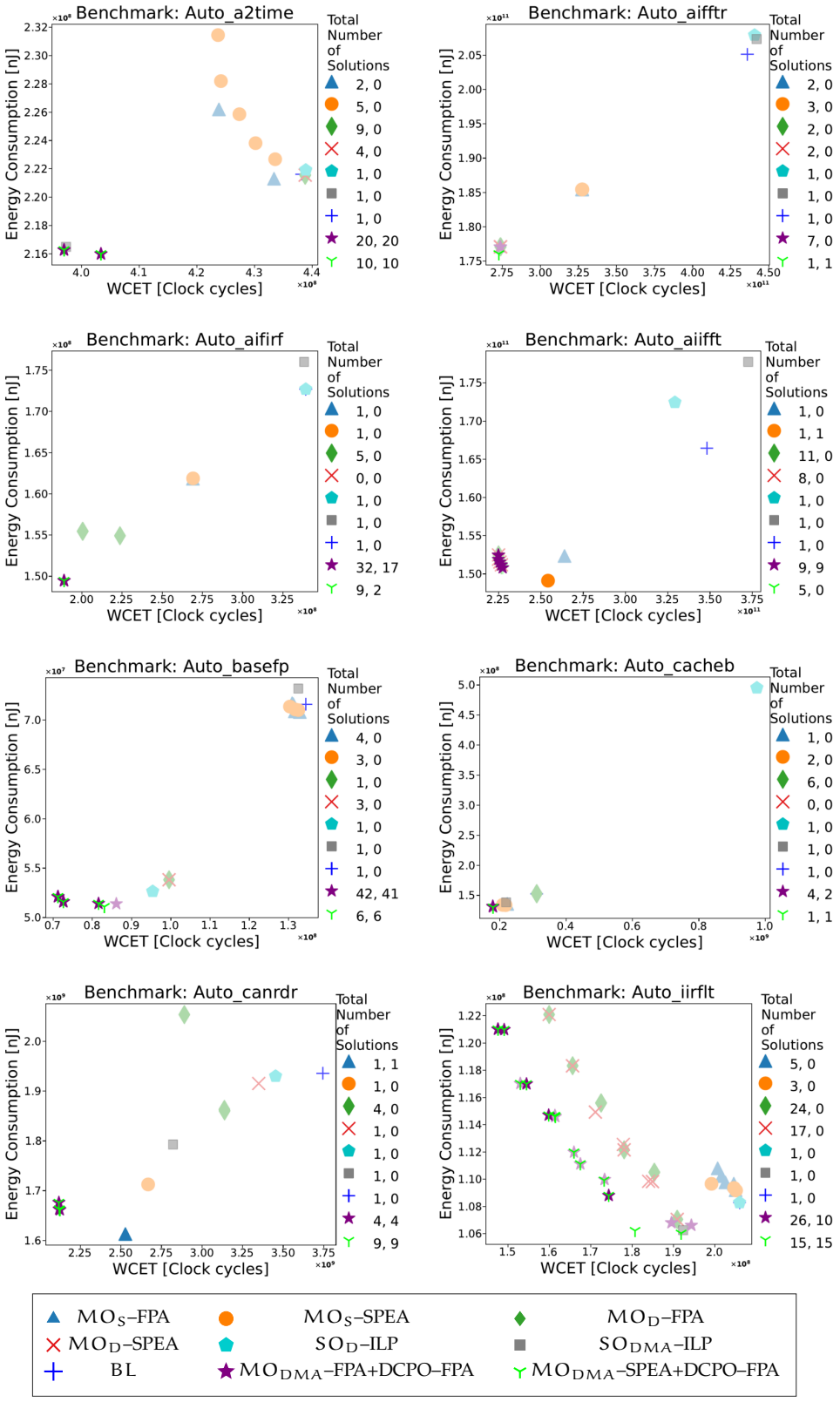


Figure E.6.: (Continued) Pareto fronts for all the benchmarks for the relative 20% SPM size and DCPO–FPA case

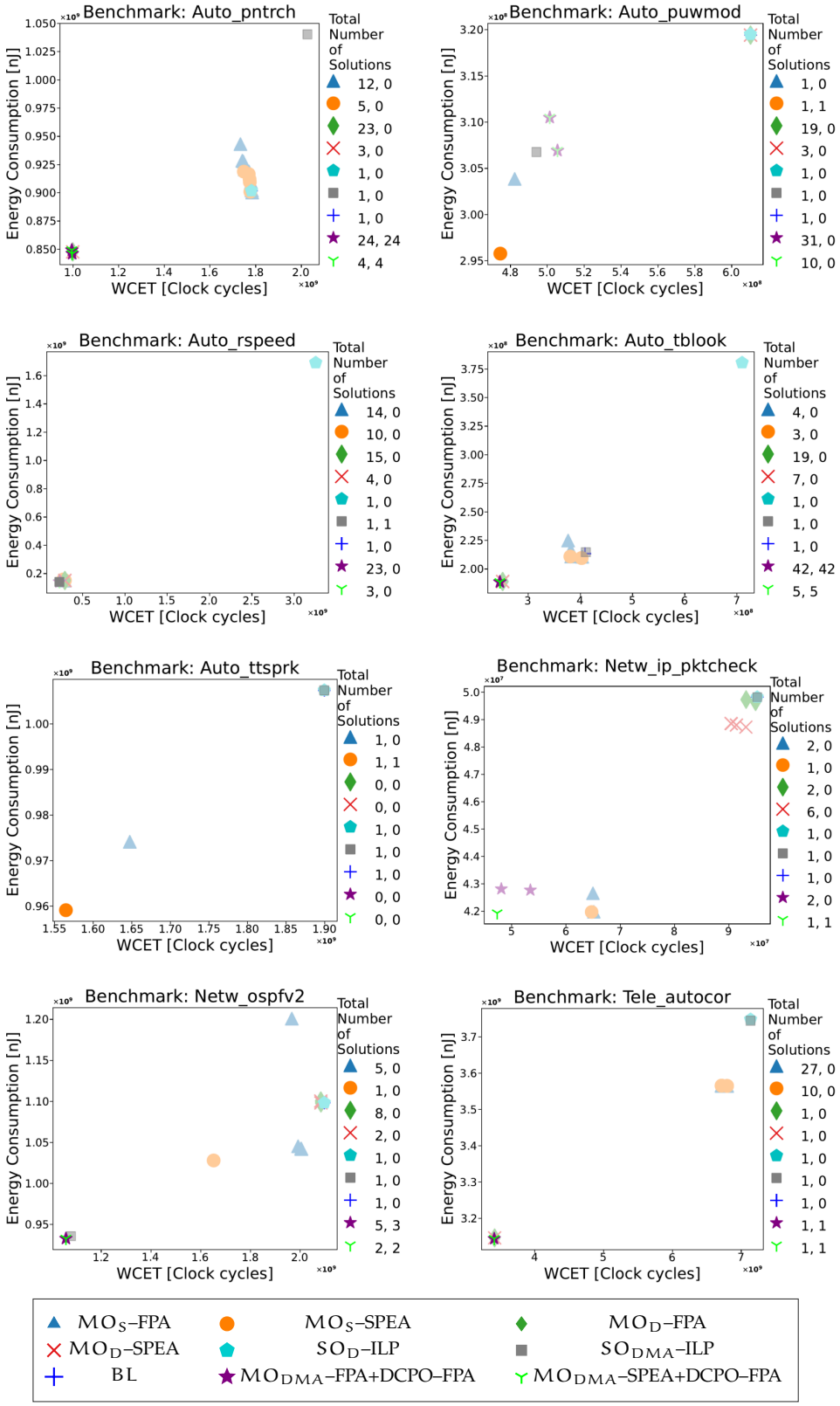


Figure E.7.: (Continued) Pareto fronts for all the benchmarks for the relative 20% SPM size and DCPO–FPA case

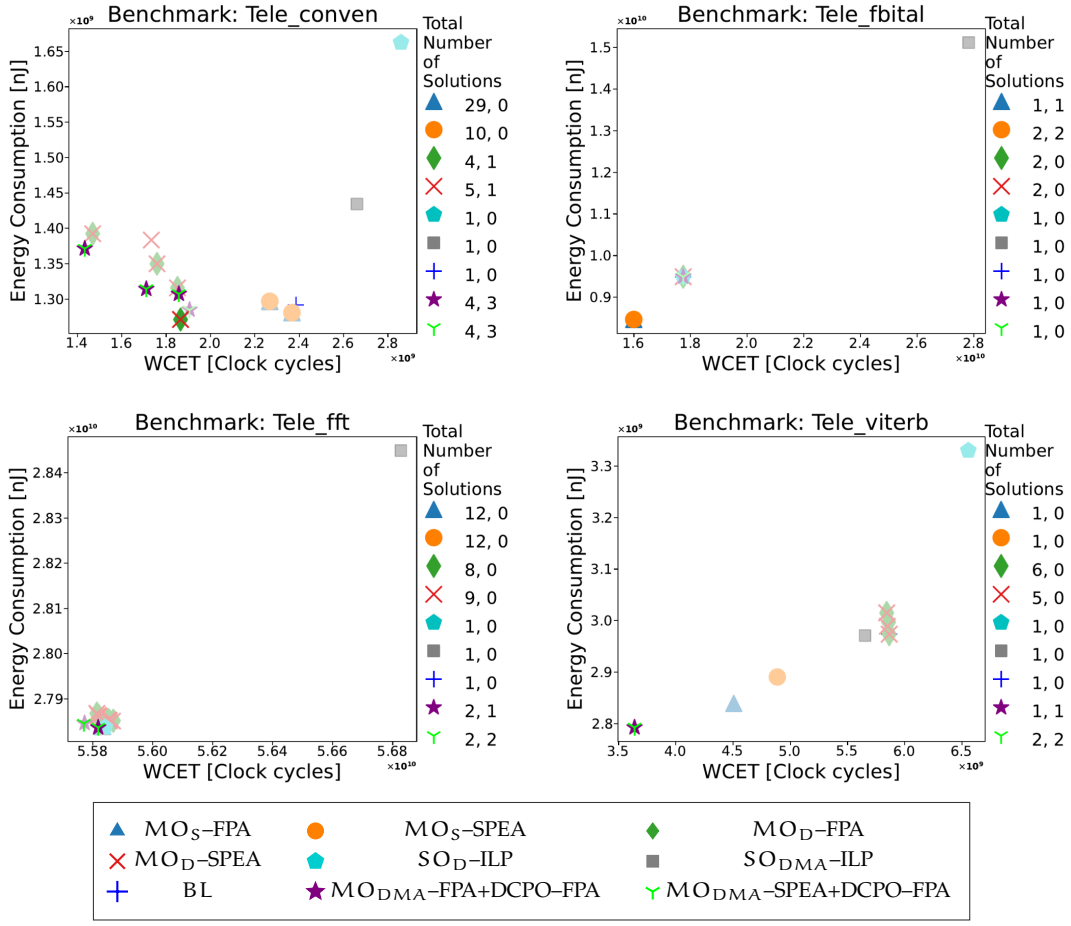


Figure E.8.: Pareto fronts for all the benchmarks using MO_D, MO_S, SO_S, SO_{DMA}, BL, and MO_{DMA} optimization approaches for the relative 40% SPM size and DCPO–FPA case

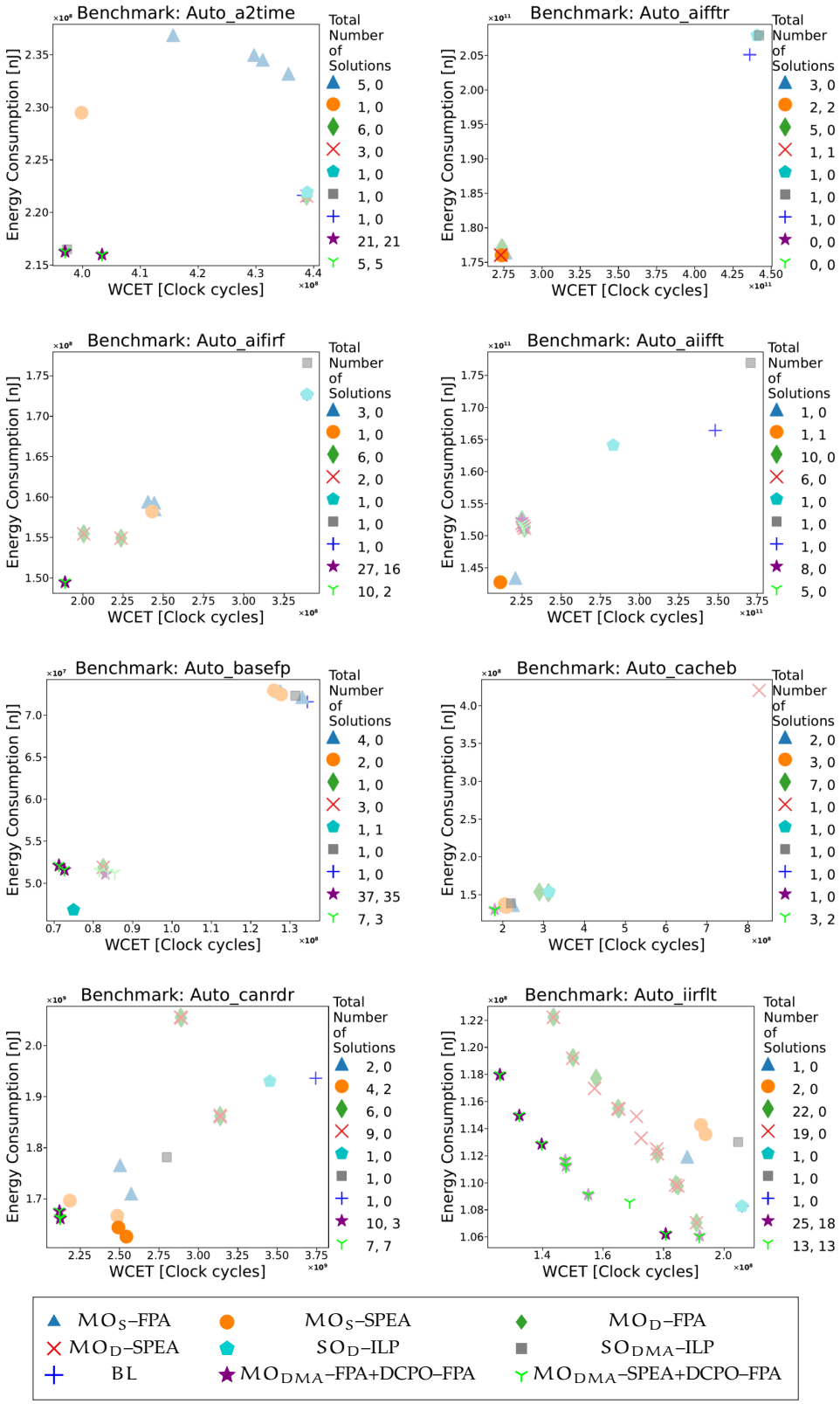


Figure E.9.: (Continued) Pareto fronts for all the benchmarks for the relative 40% SPM size and DCPO–FPA case

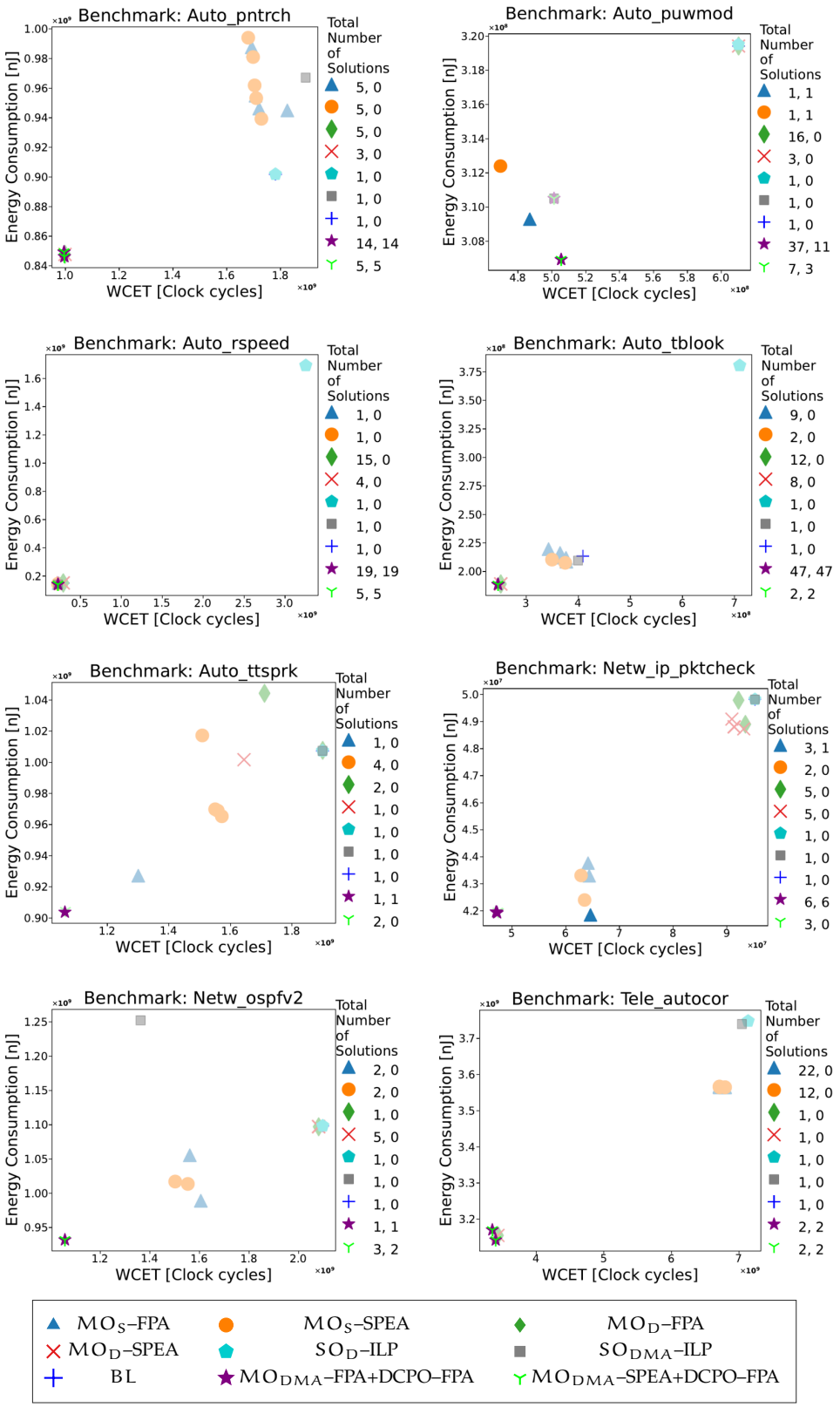


Figure E.10.: (Continued) Pareto fronts for all the benchmarks for the relative 40% SPM size and DCPO–FPA case

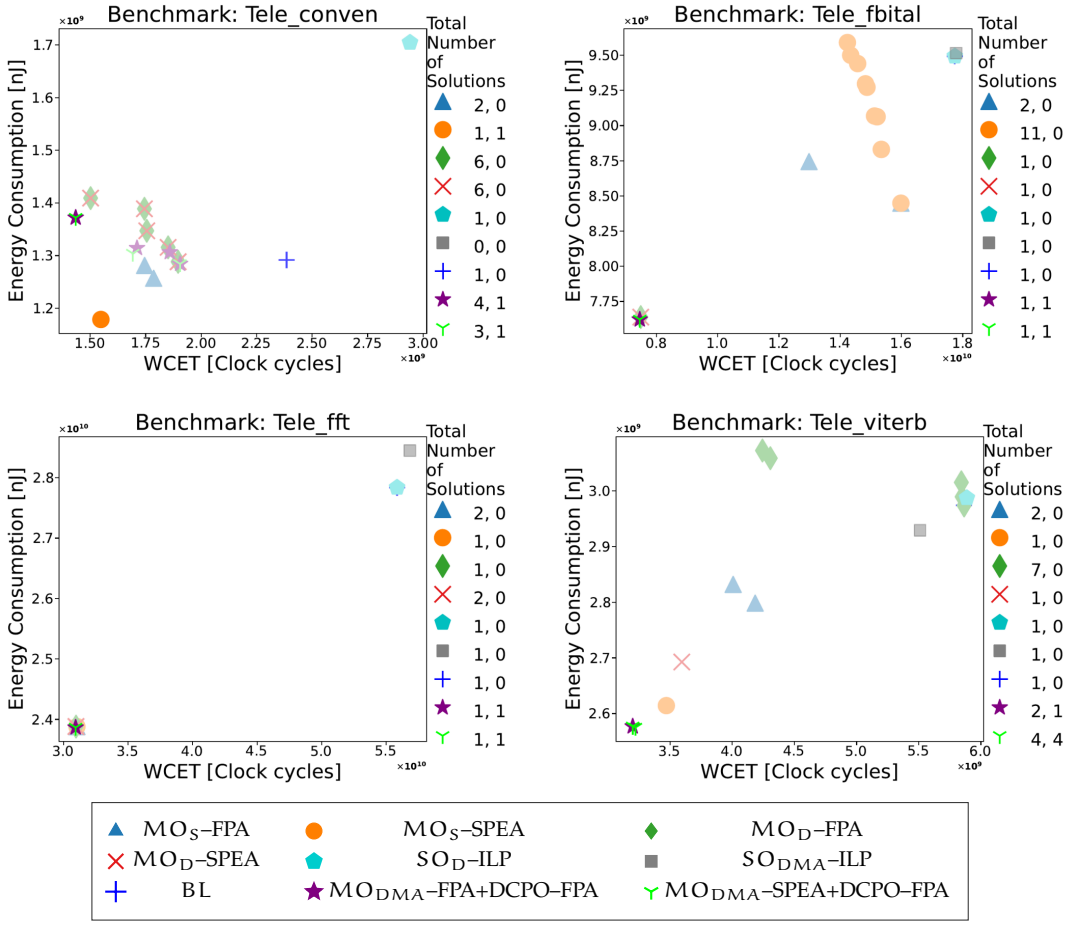


Figure E.11.: Pareto fronts for all the benchmarks using MO_D, MO_S, SO_S, SO_{DMA}, BL, and MO_{DMA} optimization approaches for the relative 80% SPM size and DCPO–FPA case

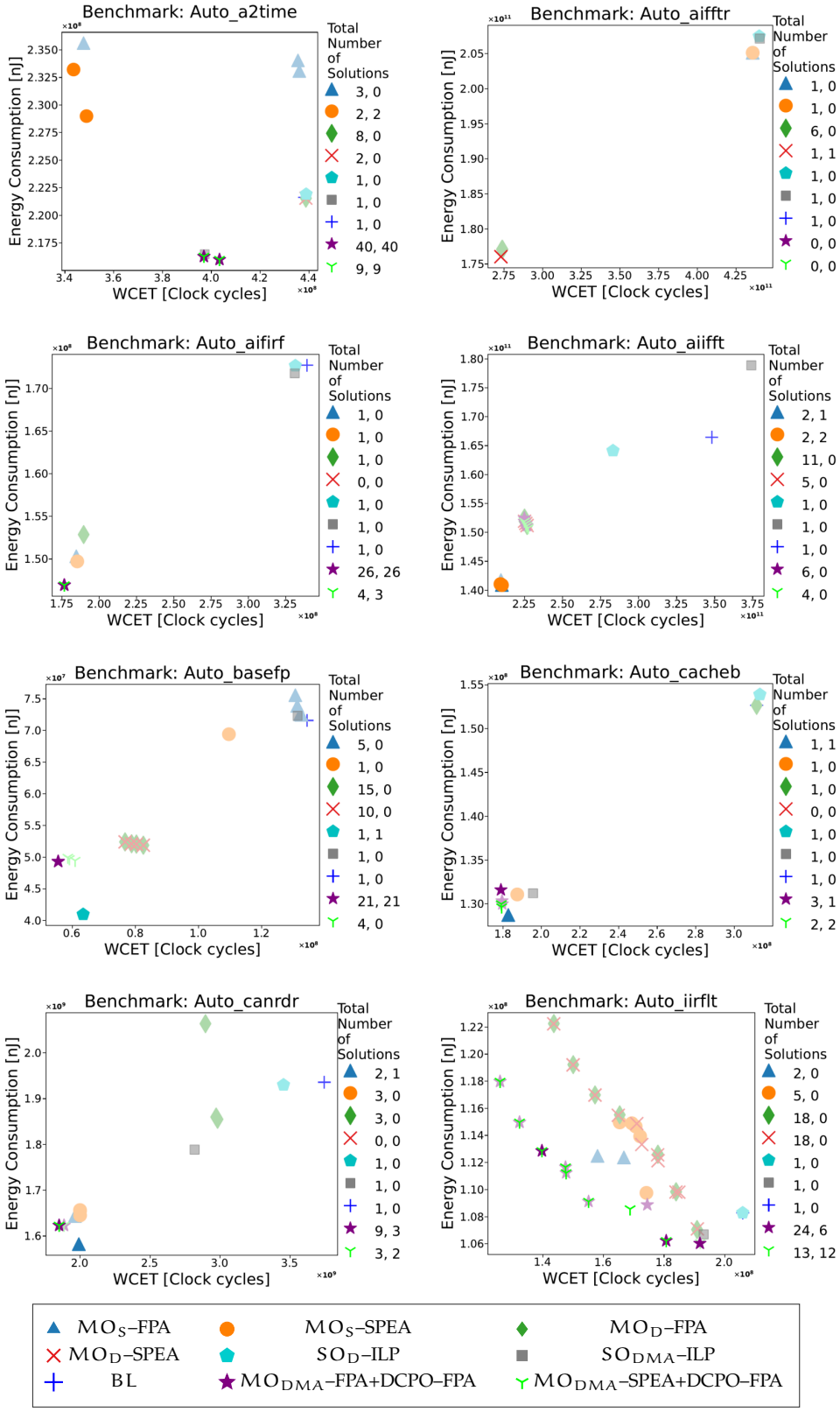


Figure E.12.: (Continued) Pareto fronts for all the benchmarks for the relative 80% SPM size and DCPO–FPA case

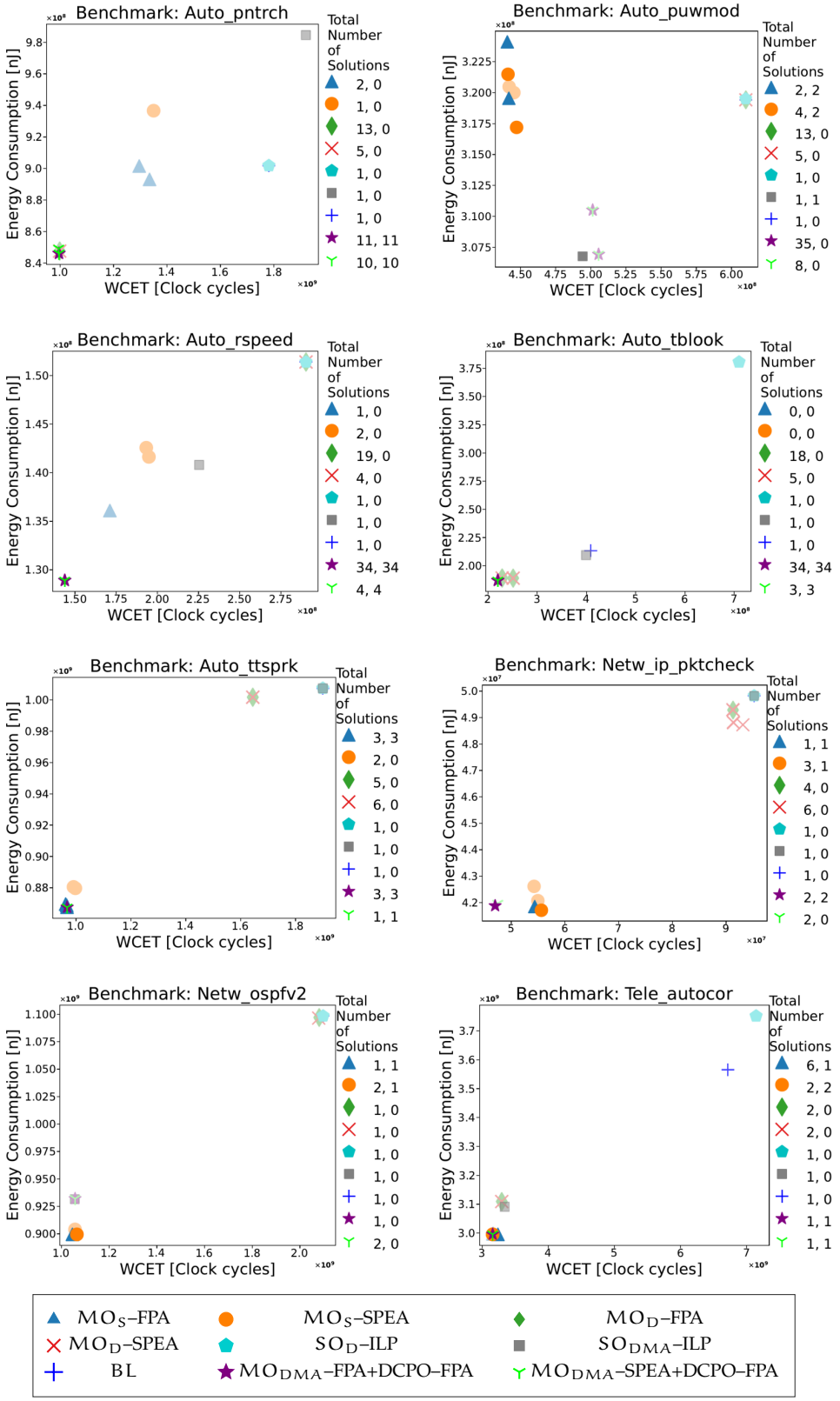
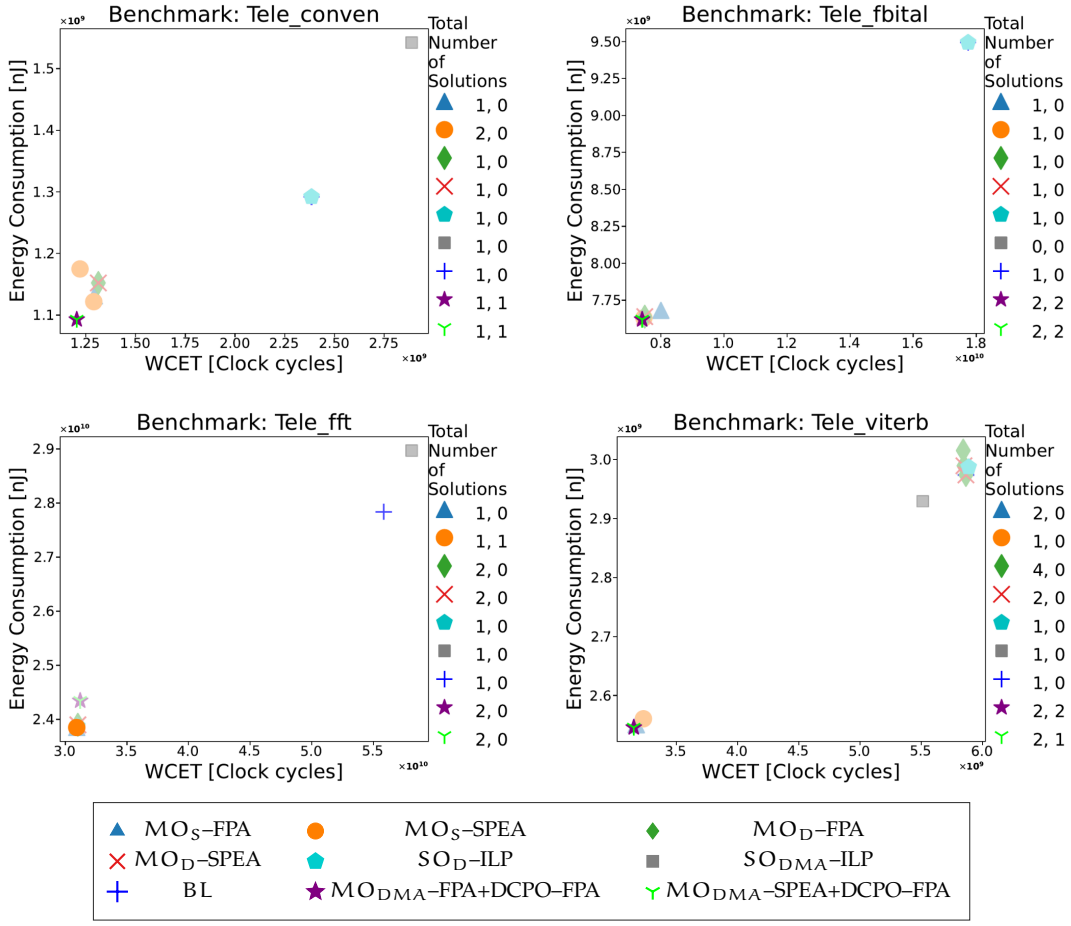


Figure E.13.: (Continued) Pareto fronts for all the benchmarks for the relative 80% SPM size and DCPO–FPA case



F

Pipeline-aware DMA Cost Model

In Chapter 9, we did not consider the effect of the multi-stage pipeline on the DMA placement cost calculations. However, in this appendix, we propose a theoretical pipeline-aware DMA cost model for an underlying 3-stage pipeline architecture. This appendix focuses exclusively on the theoretical model. As the practical implementation of this model within WCC, as well as an extensive comparison with the pipeline-unaware model from Section 9.3.2, is still missing, we did not include it in the main part of this thesis.

Let us first define some terms used to calculate the basic block-level placement cost. Let b_p^j be the j^{th} basic block in the p^{th} function. $\mathbf{N}^j$ is the total number of instructions in basic block b_p^j. $\mathbf{N}_{lF}^j$ is the number of load instructions loading from the Flash, and $\mathbf{N}_{lS}^j$ is the number of load instructions loading from the SPM. $\mathbf{N}_{sF}^j$ is the number of store instructions storing words in the Flash, and $\mathbf{N}_{sS}^j$ is the number of store instructions storing words in the SPM. The 3-stage pipeline performs instruction fetching, decoding, and execution in three phases. The fetching stage depends on the access latency of the memory in which the basic block is placed. As mentioned in Subsection 9.3.2, aT_F and aT_S are the access latencies of the Flash and SPM, respectively. t_D is the number of cycles required to decode an instruction, and t_E is the number of cycles required to execute an instruction that does not access memories. Since load and store instructions access memory during their execution phase, the number of cycles required for their execution phase is equal to the corresponding memory access latency. A basic block can be executed either from the Flash or from the SPM. Let ζ_p^j be the indicator specifying whether the basic block b_p^j is placed in the Flash or SPM.

$$\zeta_p^j = \begin{cases} 1, & \text{if } b_p^j \text{ is in SPM} \\ 0, & \text{if } b_p^j \text{ is in Flash} \end{cases} \tag{F.1}$$

In this appendix, we first present all the required terms and concepts for the pipeline-aware DMA cost model. Similar to Subsections 9.3.2.2 and 9.3.2.3, we then compute the basic block-level placement cost in arbitration mode ($t_{(p,m,a)}^j$) and the total execution time cost in arbitration mode ($\mathcal{T}'(y)$) using the pipeline-aware model. Similar to the pipeline-unaware model proposed in

Figure F.1.: Exemplary basic block code with three instructions, where the basic block is in the Flash memory. We assume that the basic block has one load and one store instruction, and the access latency for the Flash memory is 3 cycles ($aT_F = 3$).

Cycles	$Inst_1$-Ld	$Inst_2$-St	$Inst_3$
1	F_1		
2	F_1		
3	F_1		
4	D_1	F_2	
5	$E_1 - Ld$	Stall	
6	$E_1 - Ld$	Stall	
7	$E_1 - Ld$	Stall	
8		F_2	
9		F_2	
10		D_2	F_3
11		$E_2 - St$	Stall
12		$E_2 - St$	Stall
13		$E_2 - St$	Stall
14			F_3
15			F_3
16			D_3
17			E_3

Subsection 9.3.2, the pipeline-aware cost model introduced in this appendix can also be used to compute the $\mathcal{T}'(y)$ cost when solving the DCPO problem.

F.1 Basic Block-level Net Execution Time

Figure F.1 shows an exemplary basic block with three instructions ($\mathbf{N}^j = 3$) placed in the Flash memory. $Inst_1$ is a load instruction loading from the Flash ($\mathbf{N}^j_{lF} = 1$), $Inst_2$ is a store instruction storing to the Flash ($\mathbf{N}^j_{sF} = 1$), and $Inst_3$ is a normal instruction. F_1, F_2, and F_3 indicate their fetching stage, D_1, D_2, and D_3 shows their decode stage, and E_1, E_2, and E_3 indicate their execute phase. In this example, we assume $t_D = 1$ cycle and $t_E = 1$ cycle. As we can see, while executing the load instruction ($E_1 - Ld$ stage), the fetching of $Inst_2 - St$ is stalled; a similar phenomenon occurs between $Inst_2 - St$ and $Inst_3$.

For each basic block, we calculate the net execution time denoted by $\mathcal{B}^j_p$. The net execution time of a basic block depends on the memory in which the basic block is placed. Let $\mathcal{B}^j_{pF}$ be the net execution time of the basic block b^j_p, where b^j_p is placed in the Flash memory.

Figure F.2.: Exemplary basic block code with three instructions, where the basic block is in SPM. We assume that the basic block has one load and one store instruction, and the access latency for the SPM is 1 cycle ($aT_S = 1$).

Cycles	$\texttt{Inst}_1$-Ld	$\texttt{Inst}_2$-St	$\texttt{Inst}_3$
1	F_1		
2	D_1	F_2	
3	$E_1 - Ld$	D_2	Stall
4		$E_2 - St$	Stall
5			F_3
6			D_3
7			E_3

$$\mathcal{B}^j_{pF} = t_D + t_E + (\mathbf{N}^j + \mathbf{N}^j_{lF} + \mathbf{N}^j_{sF})aT_F \tag{F.2}$$

We need to add $\mathbf{N}^j_{lF}$ and $\mathbf{N}^j_{sF}$ terms in Equation F.2 to compensate for the extra stalls caused by the load and store executions. If we consider the basic block example from Figure F.1, the net execution time of the basic block is $\mathcal{B}^j_{pF} = 1 + 1 + (3 + 1 + 1) * 3 = 17$ cycles. Therefore, the net execution time calculated using Equation F.2 matches the net execution time shown in Figure F.1 in this case.

Figure F.2 shows the same net execution of the same basic block from Figure F.1; however, the basic block is placed in the SPM instead of the Flash memory. Furthermore, $\mathtt{Inst}_1$ is a load instruction loading from SPM ($\mathbf{N}^j_{lS} = 1$), and $\mathtt{Inst}_2$ is a store instruction storing to SPM ($\mathbf{N}^j_{sS} = 1$). Let $\mathcal{B}^j_{pS}$ be the net execution time of the basic block b^j_p, where b^j_p is placed in the SPM.

$$\mathcal{B}^j_{pS} = t_D + t_E + (\mathbf{N}^j + \mathbf{N}^j_{lS} + \mathbf{N}^j_{sS})aT_S + (\mathbf{N}^j_{lF} + \mathbf{N}^j_{sF})(aT_F - aT_S) \tag{F.3}$$

We need to add the $(\mathbf{N}^j_{lF} + \mathbf{N}^j_{sF})(aT_F - aT_S)$ term to Equation F.3 in case a load or store instruction in SPM is loading or storing to the Flash memory. This term compensates for the difference between the access latencies of the two memories. The number of cycles needed by the execution phase of the load/store instruction is equal to the respective memory's access latency, where the loading or storing action takes place. Therefore, if the load/store instruction is in the SPM and the load/store instruction is loading/storing to the Flash memory, the number of cycles required by the execution phase of the load/store instruction is equal to Flash's access latency (aT_F).

As αT_F is always greater than αT_S, we do not need to add such an extra term to Equation F.2. If we consider the basic block example from Figure F.2, the net execution time of the basic block is $\mathcal{B}^j_{pS} = 1 + 1 + (3 + 1 + 1) * 1 + (0 + 0)(3 - 1) = 7$ cycles. Therefore, the net execution time calculated using Equation F.3 matches the net execution time shown in Figure F.2 in this case.

Finally, we define the net execution time $\mathcal{B}^j_p$ for basic block b^j_p as the combination of both the Flash and SPM cases using Equations F.1, F.2, and F.3 as follows:

$$\mathcal{B}^j_p = \neg\zeta^j_p \mathcal{B}^j_{pF} + \zeta^j_p \mathcal{B}^j_{pS} \tag{F.4}$$

F.2 Basic Block-level DMA-aware Total Execution Time

Next, we want to calculate the total execution time of the basic block b^j_p when the DMA controller and the CPU are working in parallel. Figure F.3 expands the basic block example from Figure F.1. In Figure F.3, the DMA controller and the CPU are sharing the bus matrix. As we assume that the bus matrix employs round-robin arbitration (cf. Section 2.4.3), we need to ensure that the CPU is granted access at least half the time. The contention between the DMA controller and CPU occurs only when they try to access the same memory. The "DMA Access" column in the figure shows when the DMA controller is accessing which memory, i.e., the DMA controller accessing the Flash is represented by DMA_F, and the DMA controller accessing the SPM is shown by DMA_S. As the basic block is placed in the Flash memory in the example, the "CPU Access" column says "No" when the DMA controller is accessing the Flash, and the execution of the instructions is stalled.

Figure F.4 shows a similar expansion of the basic block example from Figure F.2, where the basic block is placed in the SPM. In this example, the "CPU Access" column says "Yes" when the DMA is accessing the Flash and the CPU is accessing the SPM. Let $\mathcal{DE}^j_p$ be the total execution time of the basic block b^j_p calculated when the DMA controller and the CPU are working in parallel. We use these two examples to calculate the $\mathcal{DE}^j_p$ of the basic block b^j_p.

Let $\mathcal{DE}^j_{pF}$ be the $\mathcal{DE}^j_p$ of the basic block b^j_p when b^j_p is placed in the Flash (the case presented in Figure F.3). We use $\mathcal{B}^j_{pF}$ from Equation F.2 to calculate $\mathcal{DE}^j_{pF}$. However, the $\mathcal{DE}^j_{pF}$ value for the basic block b^j_p is higher than its $\mathcal{B}^j_{pF}$ as the DMA controller and CPU are competing for the bus matrix in this case. The number of times the CPU and the DMA controller clash is equivalent to the number of times both try to access the memory. Consequently, we need to add an extra "$(\mathbf{N}^j + \mathbf{N}^j_{lF} + \mathbf{N}^j_{sF})\alpha T_F$" term to account for all the extra stalls caused due to the contention between the DMA controller and the CPU, i.e.,

Figure F.3.: Example basic block from Figure F.1, when the DMA controller and the CPU are working in parallel.

Cycles	DMA Access	CPU Access	$\texttt{Inst}_1$-Ld	$\texttt{Inst}_2$-St	$\texttt{Inst}_3$
1	DMA_F	No	Stall		
2	DMA_F	No	Stall		
3	DMA_F	No	Stall		
4	DMA_S	Yes	F_1		
5		Yes	F_1		
6		Yes	F_1		
7	DMA_F	No	D_1		
8	DMA_F	No	Stall		
9	DMA_F	No	Stall		
10	DMA_S	Yes	$E_1 - Ld$		
11		Yes	$E_1 - Ld$		
12		Yes	$E_1 - Ld$		
13	DMA_F	No		Stall	
14	DMA_F	No		Stall	
15	DMA_F	No		Stall	
16	DMA_S	Yes		F_2	
17		Yes		F_2	
18		Yes		F_2	
19	DMA_F	No		D_2	
20	DMA_F	No		Stall	
21	DMA_F	No		Stall	
22	DMA_S	Yes		$E_2 - St$	
23		Yes		$E_2 - St$	
24		Yes		$E_2 - St$	
25	DMA_F	No			Stall
26	DMA_F	No			Stall
27	DMA_F	No			Stall
28	DMA_S	Yes			F_3
29		Yes			F_3
30		Yes			F_3
31	DMA_F	No			D_3
32	DMA_F	No			E_3

$$\mathcal{DE}^{j}_{pF} = 2\mathcal{B}^{j}_{pF} - (t_D + t_E) \tag{F.5}$$

From the example in Figure F.1, we know that $\mathcal{B}^{j}_{pF} = 17$. Therefore, using Equation F.5, the total execution time $\mathcal{DE}^{j}_{pF}$ for the example described in

Figure F.4.: Example basic block from Figure F.2, when the DMA controller and CPU are working in parallel.

Cycles	DMA Access	CPU Access	$\texttt{Inst}_1$-Ld	$\texttt{Inst}_2$-St	$\texttt{Inst}_3$
1	DMA_F	Yes	F_1		
2	DMA_F	Yes	D_1	F_2	
3	DMA_F	Yes	$E_1 - Ld$	D_2	
4	DMA_S	No		Stall	
5	DMA_F	Yes		$E_2 - St$	Stall
6	DMA_F	Yes			F_3
7	DMA_F	Yes			D_3
8	DMA_S	No			E_3

Figure F.3 is $\mathcal{DE}^j_{pF} = 2 * 17 - (1 + 1) = 32$, which is equivalent to the total execution time of the basic block from Figure F.3.

Next, let $\mathcal{DE}^j_{pS}$ be the $\mathcal{DE}^j_p$ of the basic block b^j_p when b^j_p is placed in the SPM (the case presented in Figure F.4). We use $\mathcal{B}^j_{pS}$ from Equation F.3 to calculate $\mathcal{DE}^j_{pS}$, i.e.,

$$\mathcal{DE}^j_{pS} = \mathcal{B}^j_{pS} + (\mathbf{N}^j_{lF} + \mathbf{N}^j_{sF})(aT_F - aT_S) + \left\lfloor \frac{\mathcal{B}^j_{pS}}{aT_F + aT_S} \right\rfloor \tag{F.6}$$

Similar to Equation F.3, we need to add an extra "$(\mathbf{N}^j_{lF} + \mathbf{N}^j_{sF})(aT_F - aT_S)$" to Equation F.6 to account for any extra stalls that might be caused due to the load or store instructions that are loading or storing to the Flash from SPM. Moreover, we also add another extra "$\left\lfloor \frac{\mathcal{B}^j_{pS}}{aT_F+aT_S} \right\rfloor$" term to account for extra stalls that might be caused due to contention when both the DMA controller and the CPU are trying to access the SPM. In Figure F.4, one such stall is caused at cycle 5.

From the example in Figure F.2, we know that $\mathcal{B}^j_{pS} = 7$. Therefore, using Equation F.6, the total execution time $\mathcal{DE}^j_{pS}$ for the example described in Figure F.4 is $\mathcal{DE}^j_{pS} = 7 + (0 + 0)(3 - 1) + \lfloor \frac{7}{3+1} \rfloor = 8$, which is equivalent to the total execution time of the basic block from Figure F.4. It is important to note that Equations F.2, F.3, F.5, and F.6 apply to any general basic block, not only to the example scenarios discussed in Figures F.1, F.2, F.3, and F.4.

Finally, we define the total execution time $\mathcal{DE}^j_p$ of a basic block b^j_p when the CPU and the DMA controller are working in parallel as a combination of both the Flash and the SPM cases using Equations F.1, F.5, and F.6 as follows:

$$\mathcal{DE}^j_p = \neg\zeta^j_p \mathcal{DE}^j_{pF} + \zeta^j_p \mathcal{DE}^j_{pS} \tag{F.7}$$

F.3 Basic Block-level Parallel Execution Time

In this section, we calculate the number of cycles by which the DMA transfer is effectively progressed. Let $\mathcal{D}_p^j$ be the execution time term that captures the effective DMA transfer progress during the execution of basic block b_p^j. $\mathcal{D}_p^j$ is the number of cycles for which the DMA controller and the CPU run in parallel for b_p^j.

$$\mathcal{D}_p^j = \begin{cases} \mathcal{DE}_p^j, & \text{if } \zeta_p^j == 1 \wedge (\mathbf{N}_{lF}^j + \mathbf{N}_{sF}^j) == 0 \\ \mathcal{DE}_p^j - \left\lfloor \frac{\mathcal{DE}_p^j}{2aT_F} \right\rfloor (aT_F - aT_S), & \text{if } \zeta_p^j == 0 \vee (\mathbf{N}_{lF}^j + \mathbf{N}_{sF}^j) \neq 0 \end{cases} \tag{F.8}$$

Depending on the value of ζ_p^j and the presence of the load and store instructions in the basic block, the corresponding case in Equation F.8 decides the value of $\mathcal{D}_p^j$ as follows:

Case 1 $\mathcal{D}_p^j = \mathcal{DE}_p^j$, if $\zeta_p^j == 1 \wedge (\mathbf{N}_{lF}^j + \mathbf{N}_{sF}^j) == 0$ **:**

The $\mathcal{D}_p^j$ value is equal to $\mathcal{DE}_p^j$ if the basic block b_p^j is placed in the SPM and b_p^j has no load and store instructions, which are loading or storing to the Flash from SPM. This case is equivalent to the example described in Figure F.4. Therefore, in this example, the execution time for which the DMA controller and the CPU run in parallel is $\mathcal{D}_p^j = \mathcal{DE}_p^j = 8$ cycles (evident from Figure F.4).

Case 2 $\mathcal{D}_p^j = \mathcal{DE}_p^j - \left\lfloor \frac{\mathcal{DE}_p^j}{2aT_F} \right\rfloor (aT_F - aT_S)$, if $\zeta_p^j == 0 \vee (\mathbf{N}_{lF}^j + \mathbf{N}_{sF}^j) \neq 0$ **:**

This case is used if the basic block is placed in the Flash memory or the basic block has any load or store instructions that are loading or storing to the Flash memory, either from SPM or the Flash itself. It implies that due to the contention between the CPU and the DMA controller, the execution time term $\mathcal{D}_p^j$ of b_p^j is less than the total execution time $\mathcal{DE}_p^j$ in this case. This scenario is equivalent to the example described in Figure F.3. Consequently, in this example, the execution time for which the DMA controller and the CPU run in parallel is $\mathcal{D}_p^j = 32 - \left\lfloor \frac{32}{(2*3)} \right\rfloor (3-1) = 22$ cycles, which can be confirmed from Figure F.3 as well.

F.4 Considerations for Loops and Checks for the DMA Transfer Completions

Next, we consider the case where the DMA transfer and the execution of a loop occur in parallel. We might encounter cases where the DMA transfer is complete after a few iterations of the loop, and the remaining iterations of the

loop are executed by the CPU without interruptions from the DMA controller. We consider such cases by estimating the execution count of a basic block for which the DMA controller and the CPU might run in parallel. Let $\mathcal{EC}_p^j$ be the execution count of the basic block b_p^j for which the execution of b_p^j and the DMA transfer occur in parallel.

$$\mathcal{EC}_p^j = \begin{cases} 1, & \text{if } \forall m, b_p^j \notin \mathcal{M}_{(p,m)}, \text{ where } m \neq 1 \\ ec_p^j, & \text{if } \forall m, b_p^j \in \mathcal{M}_{(p,m)}, \text{ where } m \neq 1 \end{cases} \tag{F.9}$$

Depending on whether the basic block is part of a memory object or not, the corresponding case in Equation F.9 decides the value of $\mathcal{EC}_p^j$ as follows:

Case 1 $\mathcal{EC}_p^j = 1$, if $\forall m, b_p^j \notin \mathcal{M}_{(p,m)}$, where $m \neq 1$ **:**

If the basic block b_p^j is not a part of any loop memory object, it cannot be a part of any loop. Therefore, we can clearly assume that the execution count $\mathcal{EC}_p^j$ of b_p^j is equal to one. $m == 1$ implies that the memory object $\mathcal{M}_{(p,m)}$ is a function memory object.

Case 2 $\mathcal{EC}_p^j = ec_p^j$, if $\forall m, b_p^j \in \mathcal{M}_{(p,m)}$, where $m \neq 1$ **:**

This case is evoked if b_p^j is part of a loop, and ec_p^j is the execution count of b_p^j for which the DMA controller and the execution of b_p^j occur in parallel.

To calculate the value of ec_p^j, we need to first calculate the following two terms: $\mathcal{DC}_{(p,m,a)}^j$ and $\mathcal{D}_{(p,m')}^{loop}$. Let $\mathcal{DC}_{(p,m,a)}^j$ be the term that checks if the DMA transfer will be complete while executing the current basic block b_p^j.

$$\mathcal{DC}_{(p,m,a)}^j = t_s(\beta_{(p,m,a)}) - \sum_{j'=(r+1)}^{j} \mathcal{EC}_p^{j'} \mathcal{D}_p^{j'} \tag{F.10}$$

Equation F.10 does a similar job as Equation 9.5, where the value of $\mathcal{DC}_{(p,m,a)}^j$ tells us how many cycles worth of DMA transfer is already complete.

Let $\mathcal{D}_{(p,m')}^{loop}$ be the execution time for which the DMA controller and the CPU run in parallel for one iteration of a loop memory object $\mathcal{M}_{(p,m')}$, where $m' \neq m$, i.e., the loop $\mathcal{M}_{(p,m')}$ is different than the memory object $\mathcal{M}_{(p,m)}$ that is being allocated using the DMA controller. If the loop consists of nested loops within it, the total execution time of the nested loop for which the DMA controller and the CPU will run in parallel is also considered, i.e.,

$$\mathcal{D}^{loop}_{(p,m')} = \sum_{k=1}^{\left|\mathcal{M}_{(p,m')}\right|} \mathcal{D}^{k}_{p} + \sum_{\substack{\forall \mathcal{M}_{(p,m'')} | \mathcal{M}_{(p,m'')} \\ \text{is nested in } \mathcal{M}_{(p,m')}}} \mathcal{EC}^{m''^{max}}_{p} \sum_{\forall k''=1}^{\left|\mathcal{M}_{(p,m'')}\right|} \mathcal{D}^{k''}_{p} \tag{F.11}$$

$\mathcal{EC}^{m''^{max}}_{p}$ is the maximum loop bound of loop $\mathcal{M}_{(p,m'')}$, which is the nested loop within loop $\mathcal{M}_{(p,m')}$. $\mathcal{D}^{k}_{p}$ from Equation F.8 captures the execution time (in cycles) by which the DMA transfer is effectively progressed during the execution of the basic block b^{k}_{p}. Consequently, the first term, "$\sum_{k=1}^{\left|\mathcal{M}_{(p,m')}\right|} \mathcal{D}^{k}_{p}$", in Equation F.11, calculates the summation of the number of cycles for which the CPU and the DMA controller run in parallel for all basic blocks within $\mathcal{M}_{(p,m')}$.

The second term in Equation F.11 does a similar summation for the loops nested within $\mathcal{M}_{(p,m')}$. If $\mathcal{M}_{(p,m')}$ is iterated once, the loop $\mathcal{M}_{(p,m'')}$ within $\mathcal{M}_{(p,m')}$ will iterate $\mathcal{EC}^{m''^{max}}_{p}$ times. Therefore, the "$\mathcal{EC}^{m''^{max}}_{p}$" term is included in the second term of Equation F.11.

Now, using Equation F.10 and Equation F.11, let us define the execution count ec^{j}_{p} term of a basic block b^{j}_{p} used in Equation F.9 as follows:

$$ec^{j}_{p} = \max\left(1, \min\left(\left\lfloor \frac{\mathcal{DC}^{j}_{(p,m,a)}}{\mathcal{D}^{loop}_{(p,m')}} \right\rfloor, \mathcal{EC}^{j^{max}}_{p}\right)\right) \tag{F.12}$$

where $\mathcal{EC}^{j^{max}}_{p}$ is the total execution count of the basic block b^{j}_{p} within the loop. The "$\left\lfloor \frac{\mathcal{DC}^{j}_{(p,m,a)}}{\mathcal{D}^{loop}_{(p,m')}} \right\rfloor$" term in Equation F.12 provides us with the number of times the loop needs to be executed such that the remaining DMA transfer is completed during its execution. If the "$\left\lfloor \frac{\mathcal{DC}^{j}_{(p,m,a)}}{\mathcal{D}^{loop}_{(p,m')}} \right\rfloor$" term is larger than the total execution count of the loop ($\mathcal{EC}^{j^{max}}_{p}$), the min(.) operator ensures that $\mathcal{EC}^{j^{max}}_{p}$ is selected. Moreover, if the "$\left\lfloor \frac{\mathcal{DC}^{j}_{(p,m,a)}}{\mathcal{D}^{loop}_{(p,m')}} \right\rfloor$" term is smaller than $\mathcal{EC}^{j^{max}}_{p}$, the $\left\lfloor \frac{\mathcal{DC}^{j}_{(p,m,a)}}{\mathcal{D}^{loop}_{(p,m')}} \right\rfloor$ term will be selected.

Lastly, if the $\left\lfloor \frac{\mathcal{DC}^{j}_{(p,m,a)}}{\mathcal{D}^{loop}_{(p,m')}} \right\rfloor$ term is selected by the min(.) operator, it might be less than 1, i.e., the DMA transfer is complete before the complete loop

is executed once. In this case, the max(.) operator ensures that the valid execution count value "1" is selected for the ec_p^j term.

F.5 Basic Block-level Placement Cost in Arbitration Mode

Similar to Subsection 9.3.2.2, in this section, we calculate the placement cost (the extra execution time) for each basic block using all the pipeline-aware terms proposed in the previous sections. Let $t^j_{(p,m,a)}$ represent the placement cost incurred due to the basic block b^j_p between the r^{th} basic block and the *def* basic block of $\mathcal{A}_{(p,m,a)}$. As mentioned in Subsection 9.3.2.2, $r = \overline{1, B_p}$ is an index to indicate a basic block b^r_p where a DMA transfer call might potentially be placed. Moreover, B_p is the total number of basic blocks in the p^{th} function f_p. The $t^j_{(p,m,a)}$ proposed in this section is conceptually similar to the $t^j_{(p,m,a)}$ term in Equation 9.6. The only difference is that Equation 9.6 uses the pipeline-unaware model from Subsection 9.3.2 to calculate the $t^j_{(p,m,a)}$ term, and the following Equation F.13 uses the pipeline-aware model to calculate the $t^j_{(p,m,a)}$ term in this section.

$$t^j_{(p,m,a)} = \begin{cases} t_s(\beta_{(p,m,a)}), & \text{if } j = r, \\ \mathcal{EC}^j_p(\mathcal{DE}^j_p - \mathcal{B}^j_p - \mathcal{D}^j_p), & \text{if } j > r \wedge (\mathcal{DC}^j_{(p,m,a)} > 0 \vee \\ & \quad (\mathcal{DC}^{j-1}_{(p,m,a)} > 0 \wedge \mathcal{DC}^j_{(p,m,a)} \leqslant 0)) \\ 0, & \text{otherwise} \end{cases} \tag{F.13}$$

Depending on the value of j and $\mathcal{DC}^j_{(p,m,a)}$, the corresponding case in Equation F.13 decides the cost incurred by the j^{th} basic block as follows:

Case 1 $t^j_{(p,m,a)} = t_s(\beta_{(p,m,a)})$, if $j = r$ **:**

This case is the same as the first case discussed in Equation 9.6. As the DMA call $\delta_{(p,m,a)}$ is placed after the basic block b^r_p, the $(j = r)^{th}$ basic block is the first basic block where we start counting the cost for $t^j_{(p,m,a)}$. If any other DMA call ($\delta_{(p',m',a')}$) for some address object is placed immediately after $\delta_{(p,m,a)}$, the DMA transfer for $\mathcal{A}_{(p,m,a)}$ is carried out in the burst mode. In such a scenario, the $j = r$ case accounts for the total cost of transferring $\mathcal{A}_{(p,m,a)}$ in the burst mode.

Case 2 $t^j_{(p,m,a)} = \mathcal{EC}^j_p(\mathcal{DE}^j_p - \mathcal{B}^j_p - \mathcal{D}^j_p)$, if $j > r \wedge (\mathcal{DC}^j_{(p,m,a)} > 0 \vee (\mathcal{DC}^{j-1}_{(p,m,a)} > 0 \wedge \mathcal{DC}^j_{(p,m,a)} \leqslant 0))$ **:**

This case calculates the execution time cost incurred due to basic block b_p^j for $j > r$. The $\mathcal{EC}_p^j$ term from Equation F.9 tells how many iterations the CPU and the DMA controller can operate in parallel for the basic block b_p^j. The "$(\mathcal{DE}_p^j - \mathcal{B}_p^j - \mathcal{D}_p^j)$" term gives the one-time execution time cost for b_p^j. If the first case from Equation F.8 is true, the "$(\mathcal{DE}_p^j - \mathcal{B}_p^j - \mathcal{D}_p^j)$" term is equal to "$-\mathcal{B}_p^j$", implying that the effective DMA transfer during the execution of b_p^j was equal to $\mathcal{B}_p^j$ cycles. Consequently, $\mathcal{B}_p^j$ cycles will be eventually subtracted while calculating the total summation of $t_{(p,m,a)}^j$ over all the values of j (cf. Equation 9.10). Similarly, if the second case from Equation F.8 is true, the "$(\mathcal{DE}_p^j - \mathcal{B}_p^j - \mathcal{D}_p^j)$" term is equal to "$\left\lfloor \frac{\mathcal{DE}_p^j}{2aT_F} \right\rfloor (aT_F - aT_S) - \mathcal{B}_p^j$", implying that the effective DMA transfer during the execution of b_p^j was less by "$\left\lfloor \frac{\mathcal{DE}_p^j}{2aT_F} \right\rfloor (aT_F - aT_S)$" cycles.

If we consider the basic block example from Figure F.3, the one-time execution time cost will be $(\mathcal{DE}_p^j - \mathcal{B}_p^j - \mathcal{D}_p^j) = (32 - 17 - 22) = -7$. The intuition for calculating $t_{(p,m,a)}^j$ in Equations F.13 and 9.6 are similar. Therefore, the previously explained Figures 9.8, 9.9, 9.10, and 9.11 from Chapter 9 can provide a more detailed understanding.

The $t_{(p,m,a)}^j$ cost, in this case, is used for all $j > r$ cases. Moreover, we need to check if the DMA transfer will be completed while executing b_p^j, i.e., if $\mathcal{DC}_{(p,m,a)}^j > 0$, the DMA transfer is not complete during b_p^j's execution. If $\mathcal{DC}_{(p,m,a)}^j \leqslant 0$, we need to confirm if the DMA transfer was not over during the execution of the previous basic block b_p^{j-1}, i.e., $\mathcal{DC}_{(p,m,a)}^{j-1} > 0$. If one of these $\mathcal{DC}_{(p,m,a)}^j$ conditions is satisfied, we use this case to calculate the $t_{(p,m,a)}^j$ cost for b_p^j.

Case 3 $t_{(p,m,a)}^j = 0$, otherwise **:**

Lastly, if none of the previous conditions are satisfied, the $t_{(p,m,a)}^j$ cost value is set to zero. It also implies that if the DMA transfer of the address object is already over before the execution of b_p^j, then there is no need to add extra DMA-related costs.

F.6 Total Execution Time Cost in Arbitration Mode

The total execution time cost for all the DMA transfers in arbitration mode ($\mathcal{T}'(y)$) for the pipeline-aware model presented in this section has the same form as the $\mathcal{T}'(y)$ model (cf. Equation 9.11) discussed in Subsection 9.3.2.3. In

Subsection 9.3.2.3, Equation 9.6 is used to compute the $t^{j}_{(p,m,a)}$ value based on the pipeline-unaware model from Chapter 9. This value is then used to calculate $\sigma^{j}_{(p,m,a)}$ from Equation 9.10, which in turn is used to compute the $\mathcal{T}'(y)$ value.

In contrast, to compute $\mathcal{T}'(y)$ in this appendix, we use the pipeline-aware model to calculate $t^{j}_{(p,m,a)}$ using Equation F.13, while still employing Equation 9.10 to compute $\sigma^{j}_{(p,m,a)}$. Therefore, for brevity, we do not repeat the detailed explanation of the $\mathcal{T}'(y)$ calculation.

In Subsection 9.3.3, we formulated the DCPO optimization problem, which aims to minimize the $\mathcal{T}'(y)$ objective value. The DCPO optimization problem treats the calculation of $\mathcal{T}'(y)$ as a black box. Since this appendix introduces only a theoretical pipeline-aware DMA cost model, its practical implementation within the WCC compiler is left for future work, along with a comprehensive comparison against the pipeline-unaware model from Chapter 9. In future implementations of the DCPO problem, a system designer could choose either the proposed pipeline-unaware model from Chapter 9 or the pipeline-aware model proposed in this appendix to compute the $\mathcal{T}'(y)$ objective value.

G

Compression-aware Multi-Objective Dynamic SPM Allocation

Tables G.1 and G.2 present the quality indicator values for all the benchmarks from the evaluations described in Chapter 10 for MO_D, BL, and MO_C optimization runs. For any benchmark, the approach with quality indicators in bold performed better (or nondominant) than other approaches.

Figure G.1 and Figure G.2 present the Pareto fronts for all the remaining benchmarks from the evaluations described in Chapter 10 for MO_D, BL, and MO_C optimization runs.

Table G.1.: Coverage ($\mathcal{C}$) and Non-dominance Ratio (NR) metrics for MO_D, BL, and MO_C evaluation runs

	Coverage ($\mathcal{C}$)					Non-dominance Ratio (NR)				
	MO_D		BL	MO_C		MO_D		BL	MO_C	
Benchmarks	FPA	SPEA		FPA	SPEA	FPA	SPEA		FPA	SPEA
Auto_a2time	0.733	0.4	**0**	0.417	0.444	0.125	0.094	0.031	**0.437**	0.312
Auto_aifftr	1	**0**	**0**	0.667	0.4	0	0.083	0.083	0.333	**0.5**
Auto_aifirf	1	1	1	1	**0.182**	0	0	0	0	**1**
Auto_aiifft	1	1	**0**	0.667	0.222	0	0	0.077	0.385	**0.538**
Auto_basefp	1	1	1	0.812	**0.059**	0	0	0	0.158	**0.842**
Auto_cacheb	1	1	1	**0.467**	0.615	0	0	0	**0.615**	0.385
Auto_canrdr	1	1	**0**	0.333	0.458	0	0	0.045	0.364	**0.591**
Auto_iirflt	0.882	1	**0**	0.731	0.375	0.1	0	0.05	0.35	**0.5**
Auto_pntrch	**0**	0.5	**0**	0.571	0.647	0.062	0.125	0.062	**0.375**	**0.375**
Auto_puwmod	1	1	**0**	0.321	0.5	0	0	0.036	**0.679**	0.286
Auto_rspeed	1	1	1	1	**0.5**	0	0	0	0	**1**
Auto_tblook	1	1	**0**	0.741	0.143	0	0	0.038	0.269	**0.692**
Auto_ttsprk	1	0.75	**0**	1	0.28	0	0.05	0.05	0	**0.9**
Netw_ip_pktcheck	1	1	1	1	**0**	0	0	0	0	**1**
Netw_ospfv2	1	1	1	1	**0**	0	0	0	0	**1**
Tele_autocor	1	1	1	**0**	**0**	0	0	0	**0.5**	**0.5**
Tele_conven	0.714	0.625	**0**	0.8	**0**	0.2	**0.3**	0.1	0.1	**0.3**
Tele_fbital	**0**	1	1	0.75	0.5	0.143	0	0	0.286	**0.571**
Tele_fft	1	1	1	0.667	**0.143**	0	0	0	0.25	**0.75**
Tele_viterb	1	1	1	0.4	**0**	0	0	0	0.375	**0.625**

Table G.2.: Ratio of Non-dominated Solutions (RNS) and HyperVolume (HV) metrics for MO_D, BL, and MO_C evaluation runs

	Ratio of Non-dominated Solutions (RNS)					HyperVolume (HV)				
	MO_D		BL	MO_C		MO_D		BL	MO_C	
	FPA	SPEA		FPA	SPEA	FPA	SPEA		FPA	SPEA
Benchmarks										
Auto_a2time	0.267	0.6	**1**	0.583	0.556	0.1	0.098	0.112	**0.826**	**0.826**
Auto_aifftr	0	**1**	**1**	0.333	0.6	0.191	0.408	0	0.681	**0.72**
Auto_aifirf	0	0	0	0	**0.818**	0.424	0.012	0	0.528	**0.762**
Auto_aiifft	0	0	**1**	0.333	0.778	0.236	0.275	0	0.575	**0.671**
Auto_basefp	0	0	0	0.187	**0.941**	0.108	0.098	0	**0.606**	0.597
Auto_cacheb	0	0	0	**0.533**	0.385	0.004	0	0.448	**0.914**	0.9
Auto_canrdr	0	0	**1**	0.667	0.542	0.071	0.181	0	**0.655**	0.598
Auto_iirflt	0.118	0	**1**	0.269	0.625	0.244	0.276	0.025	0.568	**0.628**
Auto_pntrch	**1**	0.5	**1**	0.429	0.353	0.348	0.348	0.001	**0.44**	0.369
Auto_puwmod	0	0	**1**	0.679	0.5	0.242	0.068	**0.289**	0.247	0.247
Auto_rspeed	0	0	0	0	**0.5**	0	0	0	0.991	**0.993**
Auto_tblook	0	0	**1**	0.259	0.857	0.011	0.021	0	**0.578**	0.552
Auto_ttsprk	0	0.25	**1**	0	0.72	0.203	0	0.008	**0.609**	0.606
Netw_ip_pktcheck	0	0	0	0	**1**	0.43	0.143	0.53	0.656	**0.938**
Netw_ospfv2	0	0	0	0	**1**	0	0	0	0.697	**0.72**
Tele_autocor	0	0	0	**1**	**1**	0.012	0.289	0	**0.982**	**0.982**
Tele_conven	0.286	0.375	**1**	0.2	**1**	0.33	0.287	0	0.423	**0.536**
Tele_fbital	**1**	0	0	0.25	0.5	0.379	0.018	0	**0.725**	**0.725**
Tele_fft	0	0	0	0.333	**0.857**	0.05	0.006	0	0.816	**0.832**
Tele_viterb	0	0	0	0.6	**1**	0	0	0	**0.97**	0.904

Figure G.1.: Pareto fronts for the remaining benchmarks using MO_D, BL, and MO_C optimization approaches

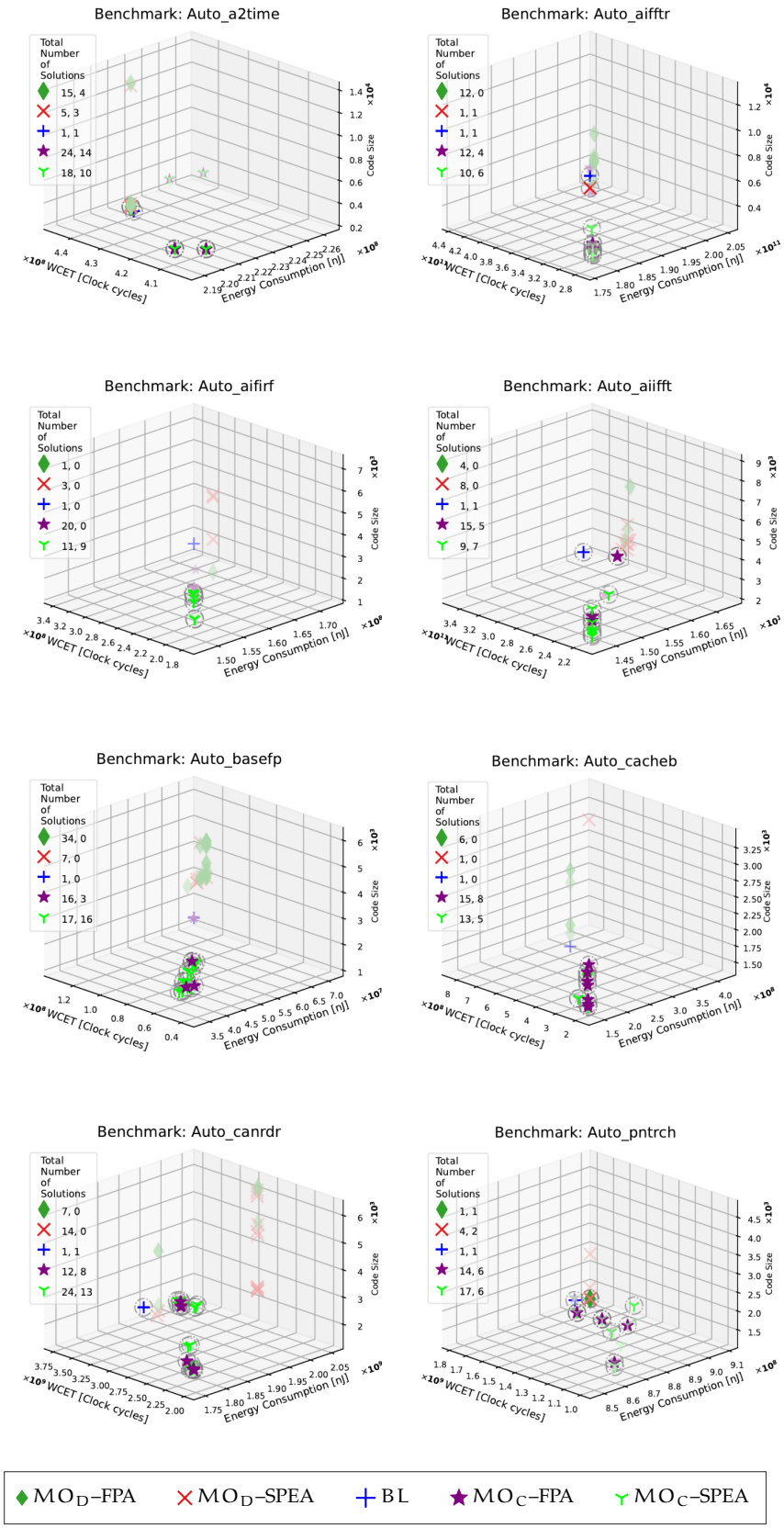


Figure G.2.: (Continued) Pareto fronts for the remaining benchmarks

