Real-Time Systems
https://doi.org/10.1007/s11241-025-09436-w

Compiler-level DMA-aware multi-objective dynamic SPM allocation

Shashank Jadhav[1] · **Heiko Falk**[1]

Accepted: 14 March 2025

Abstract

Real-time embedded systems need to meet timing and energy constraints to avoid potential disasters. Compiler-level ScratchPad Memory (SPM) allocation can be used to optimize a program's Worst-Case Execution Time (WCET) and energy consumption. However, static allocation is limited by SPM size constraints. Dynamic SPM allocation resolves this by allocating code to SPM during runtime, but copying code using the CPU increases WCET and energy consumption. To address this, we integrate a Direct Memory Access (DMA) model and DMA analysis at the compiler level and propose a single-objective DMA Call Placement Optimization (*DCPO*). In this paper, we consider functions and loops as dynamic allocation candidates. *DCPO* finds appropriate places within the code to place DMA transfer calls such that the DMA controller and the CPU run parallelly—minimizing the total execution time required by the DMA controller for dynamic allocation of functions and loops during runtime. Additionally, we propose a compiler-level DMA-aware multi-objective dynamic SPM allocation that uses *DCPO* and simultaneously minimizes WCET and energy objectives, yielding Pareto optimal solutions. Comparative evaluations demonstrate the superiority of our approach over state-of-the-art multi- and single-objective optimizations.

Keywords Multi-objective optimization · Embedded systems · Dynamic SPM allocation · DMA · Metaheuristic algorithms · ILP

✉ Shashank Jadhav
shashank.jadhav@tuhh.de

Heiko Falk
heiko.falk@tuhh.de

[1] Institute Of Embedded Systems, Hamburg University of Technology, Hamburg, Germany

Published online: 09 April 2025

1 Introduction

Embedded systems incur various constraints due to the functional and temporal aspects of the system. Real-time embedded systems have hard timing constraints, and many such systems are small, mobile, and battery-operated, with limited energy to spend. As the execution paths that contribute to a program's Worst-Case Execution Time (WCET) and energy consumption can vary, WCET and energy objectives can contradict each other, i.e., minimizing energy consumption can increase a program's WCET and vice versa. Therefore, in this paper, we consider the program's WCET and energy consumption simultaneously as minimization objectives.

Embedded systems often have small but fast and energy-efficient local memories called ScratchPad Memories (SPM). In the past, compiler-level static SPM allocation-based optimizations have used SPMs to optimize either WCET or energy consumption. However, their small size limits the objects that could be allocated statically to SPM—constraining the static SPM allocation-based optimizations. On the contrary, dynamic allocation-based optimizations can circumvent the SPM size constraint.

Dynamic SPM allocation allows for the runtime movement of objects from the Flash memory to SPM. 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 overheads due to their dynamic allocation. Therefore, in this paper, 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 paper.

Interrupting the CPU to copy functions and loops from the Flash to SPM during runtime can add further WCET and energy overheads. This interference can degrade the code quality and undermine the goals of optimization. A Direct Memory Access (DMA) controller can transfer functions and loops from one memory to another, freeing the CPU to perform timing-critical executions. Therefore, we proposed a compiler-level DMA model and an accompanying DMA analyzer in this paper. The DMA controller must finish transferring before the respective function or loop needs execution, or the CPU stalls until the 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. 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* aims to minimize the total execution time required by the DMA controller to perform dynamic allocation during runtime.

We also propose a DMA-aware dynamic SPM allocation-based multi-objective optimization (MO_{DMA}) to simultaneously minimize the program's WCET and

energy consumption. MO_{DMA} calls the *DCPO* optimization to find appropriate places for the DMA transfer calls. MO_{DMA} outputs Pareto optimal solutions that allocate functions and loops using the DMA controller during runtime and reduce the respective objective overheads. Similar to static allocation-based techniques, at compile-time, MO_{DMA} decides which functions and loops will be allocated to SPM. However, contrary to them, the contents within SPM change dynamically during runtime.

As metaheuristic algorithms are best suited to solve such multi-objective optimizations, we solve the proposed MO_{DMA} problem using two metaheuristic algorithms—the Flower Pollination Algorithm (FPA) (Yang 2012) and the Strength Pareto Evolutionary Algorithm (SPEA) (Zitzler 1999). As *DCPO* is a single-objective optimization, we solve it using Integer Linear Programming (ILP) along with FPA and SPEA. The key contributions of this paper are:

- We introduce a compiler-level DMA model and a DMA analyzer and integrate it within a 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 using industry-standard analyzers without the need for human intervention.
- We are the first to propose a DMA-aware dynamic SPM allocation-based multi-objective optimization (MO_{DMA}) to minimize the program's WCET and energy consumption and solve it using FPA and SPEA.

During evaluations, we compare the proposed MO_{DMA} with state-of-the-art single-objective ILP-based dynamic SPM allocation with and without DMA and multi-objective static and dynamic SPM allocation. We also compared MO_{DMA} with a *BaseLine* evaluation, where the whole program is executed from the Flash memory. Evaluations show that our proposed approach surpasses all the previous state-of-the-art approaches concerning the quality of the obtained solutions. 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 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, respectively.

- We achieved a 19.17% and 7.64% average reduction in WCET and energy consumption, respectively, using MO_{DMA}+*DCPO*–ILP compared to the DMA-unaware multi-objective dynamic SPM allocation that instead uses CPU for dynamic allocation.
- MO_{DMA}+*DCPO*–ILP achieved, on average, 39.57% and 11.01% reduction in WCET and energy consumption compared to the *BaseLine* evaluation.

The paper is organized as follows: Sect. 2 provides an overview of the related work. Section 3 discusses the compiler framework, within which the proposed *DCPO* and MO_{DMA} are implemented and evaluated, and defines the considered system architecture. Section 4 presents definitions and some preliminary information related to dynamic SPM allocation, *DCPO*, and MO_{DMA}. Section 5 discusses dynamic SPM allocation within the compiler. Section 6 presents the proposed compiler-level DMA model, DMA analysis, and *DCPO*. Section 7 presents the proposed MO_{DMA}. Section 8 presents the evaluation results. Lastly, Sect. 9 presents the conclusion and discussion of the future work.

2 Related work

In the past, memory allocation techniques have been explored at the compiler level, focusing on single-objective optimizations. Ishitobi et al. (2007) proposed a code placement algorithm to find optimal code layouts for SPM and cacheable and non-cacheable regions to reduce the total energy consumption of embedded systems. Balasundaram and Chenniappan (2015) proposed a metaheuristics-based optimizer to find the optimal code layout that minimizes energy consumption. Suhendra et al. (2005) presented static data SPM allocations using ILP and two different WCET-aware heuristics. Falk and Kleinsorge (2009) modified and extended this approach to perform ILP-based WCET-aware static SPM allocation for program code. Pellizzoni et al. (2011) 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, required data is preloaded into local memory or caches, and in the execution phase, the program runs using the data preloaded in the memory phase. Mancuso et al. (2014) introduced the light-PREM that automated the refactoring process needed to convert the legacy software code to PREM-compliant code. Unlike PREMs and static SPM allocations, dynamic SPM allocation allows changing and rewriting the content within SPM during runtime.

Francesco et al. (2004) exploited hardware support to reduce transfer costs to and from SPM, minimizing energy consumption, and provided a high-level programming interface for runtime scratchpad management. Qiu et al. (2017) presented an approach to reduce energy consumption for a system with a hybrid SPM architecture by deriving suitable data allocation using dynamic programming—hybrid SPM incorporates SRAM, magnetic RAM, and zero-capacitor RAM for overall performance gain. Verma and Marwedel (2007) proposed an ILP-based single-objective dynamic SPM overlay approach (SO_D) for data and instruction allocation, minimizing the program's energy consumption. Kandemir et al. (2001) presented

a compiler-level dynamic SPM management framework that uses a set of compiler optimizations and an on-chip memory space partitioning strategy to utilize the SPM as efficiently as possible. Deverge and Puaut (2007) proposed a single-objective optimization that dynamically allocates static and stack data to minimize WCET.

Udayakumaran and Barua (2003) proposed a compiler-level dynamic allocation method for global and stack data, which dynamically allocates data using DMA to reduce the runtime of a program. Soliman and Pellizzoni (2017) 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 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. (2004) used a dual-ported SPM and a dedicated DMA engine to perform runtime SPM management. Dasygenis et al. (2006) 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. (2010) 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. Kim et al. (2017) used a DMA controller for dynamic SPM allocation but in the context of a function-level single-objective ILP-based optimization (SO_{DMA}) that minimized WCET. 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 our MO_{DMA} approach, 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. Like our approach, Verma's SO_D and Kim's SO_{DMA} both consider dynamic allocation of program code, and Kim's SO_{DMA} uses DMA for dynamic allocation of functions. So, during evaluations, we compare our approach with them to evaluate the performance of single-objective optimizations against our multi-objective optimization.

Some research has been done towards compiler-level multi-objective optimization. Muts and Falk (2020) proposed a WCET-, energy-, and code size-aware multi-objective optimization that performed high-level function inlining. Lokuciejewski et al. (2011) proposed an optimization that traded Average-Case Execution Time (ACET) and WCET, and WCET and code size of a program to determine optimal compiler optimization sequences. Jadhav and Falk (2019, 2022) proposed a compiler-level multi-objective static SPM allocation (MO_S) that statically allocates basic blocks to SPM at compile time. Furthermore, Jadhav

and Falk (2023) proposed a multi-objective dynamic SPM allocation (MO_{MEM}) that dynamically allocates code fragments during runtime using CPU resources. However, their evaluations showed the quality of obtained solutions was comparable to the static allocation approach, and the use of the CPU for copying from one memory to another degraded the quality of solutions. During evaluations, we compare our proposed MO_{DMA} approach with MO_S and MO_{MEM} to determine the effectiveness of DMA-aware dynamic allocation over the past multi-objective static and dynamic allocation techniques.

Compared to ILPs, metaheuristic algorithms are better suited for handling multiple objectives simultaneously. The Flower Pollination Algorithm (FPA) (Yang 2012) is a nature-inspired metaheuristic algorithm that mimics the behavior of the pollination process in flowering plants. Strength Pareto Evolutionary Algorithm (SPEA) (Zitzler 1999) is an evolutionary algorithm—another subcategory of metaheuristic algorithms—that uses evolution-based strategies like recombination, selection, and mutation to solve an optimization problem. FPA has outperformed VEGA, NSGA-II, MODE, etc. (Yang et al. 2014), while SPEA outperformed NSGA, VEGA, etc. (Zitzler and Thiele 1999). Therefore, we chose to solve the proposed DMA-aware multi-objective dynamic SPM allocation and *DCPO* using FPA and SPEA in this paper. Moreover, ILPs are most effective for solving single-objective optimizations with multiple constraints. As the proposed *DCPO* problem fits these criteria, along with FPA and SPEA, we also formulate a *DCPO* ILP model and solve the optimization problem using an ILP solver called `Gurobi` (Gurobi Optimization Inc 2024) during evaluations.

3 Compiler infrastructure and underlying architecture

The DMA-aware multi-objective dynamic SPM allocation proposed and implemented in this paper is a compiler-level optimization. We implemented this optimization in a C compiler called the WCET-aware C Compiler (WCC) (Falk and Lokuciejewski 2010). Figure 1 shows a schematic of the internal structure of the WCC compiler. The WCC compiler supports TriCore TC1796 and TC1797 processors from Infineon and Cortex-M0 and ARM7TDMI processors from ARM. WCC consists of components that resemble a typical optimizing compiler, i.e., a parser, a high-level C-like Intermediate Representation (IR), a code selector, a low-level assembly-level IR, and a code generator. However, we implemented the proposed MO_{DMA} within WCC due to its ability to perform WCET and energy analyses during compilation.

A static WCET analyzer tool called `aiT` (AbsInt Angewandte Informatik, GmbH 2024) and an energy analysis tool called `EnergyAnalyzer` (TeamPlay Consortium 2020) are integrated tightly within WCC. WCET and energy analysis are carried out after generating the low-level IR within the WCC compiler (cf. Fig. 1). Both `aiT` and `EnergyAnalyzer` are industry-standard analyzers, and WCC uses them to perform WCET and energy analyses while solving the optimizations during evaluations. Therefore, we are able to perform the optimization and respective WCET and energy analysis during compilation without any human intervention.

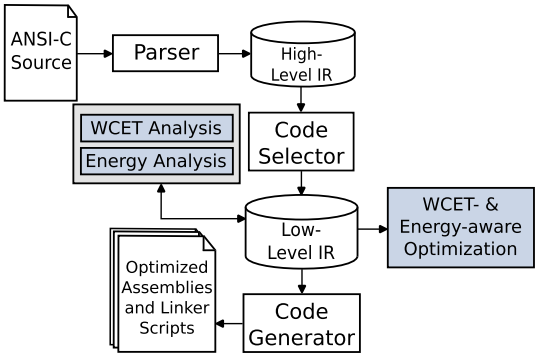


Fig. 1 Compiler's internal framework

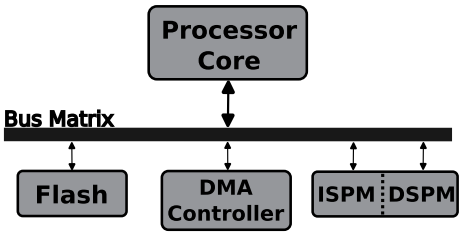


Fig. 2 Exemplary memory hierarchy with DMA

We perform MO_{DMA}- and *DCPO*-related code transformations and analyses after the low-level IR stage within the WCC compiler (cf. Fig. 1). In this paper, we consider a bare-metal single-task system. The underlying architecture for dynamic SPM allocation consists of a Flash memory, an instruction SPM (*ISPM*), and a data SPM (*DSPM*). We do not consider any involvement of caches in this paper as this would inherently lead to coherency issues, and the handling of cache coherency is beyond the scope of this paper. Figure 2 shows an exemplary memory hierarchy with a DMA controller used in this paper.

During evaluations, we generate code, perform analyses, and perform compiler-level MO_{DMA} optimization for ARM Cortex-M0. Figure 2 exactly represents the STM32F0xx implementation of the ARM Cortex-M0 architecture (STMicroelectronics 2022a, b). The WCET and energy analyses performed during the evaluations also consider underlying timing and energy models for the ARM Cortex-M0 architecture. The following energy model is used within the `EnergyAnalyzer` to calculate the total energy consumption for ARM Cortex-M0 architecture (Wegener et al. 2023; Nikov et al. 2022):

$$\begin{aligned} E =& 0.972565030 * \mathbb{C}_1 + 0.652871770 * \mathbb{C}_2 \\ &+ 1.031341343 * \mathbb{C}_3 + 1.037625441 * \mathbb{C}_4 \\ &+ 1.354953706 * \mathbb{C}_5 + 2.274650563 * \mathbb{C}_6 \end{aligned} \tag{1}$$

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. It is important to note that the energy model used by the `EnergyAnalyzer` does not account for the energy consumption of the DMA controller. Therefore, we are not able to account for the energy costs due to DMA arbitrations and CPU stalls during runtime. However, we estimate the energy costs incurred due to the extra memory accesses during DMA transfers in Sect. 7. We add this extra cost 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.

Typically, commercial DMA controllers are equipped with multiple DMA channels that can be programmed for simultaneous data transfers. However, in this paper, 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 overheads and pessimism in the model. As we want to model the execution time needed to perform DMA at compile time as precisely as possible in this paper, we consider a single DMA transfer at a time. Moreover, for STM32F0xx, only one DMA channel can be programmed to perform memory-to-memory transfer at a time. Memory-to-memory DMA transfer requires three steps: arbitration for bus access, address computation, and data transfer.

In arbitration for the bus access step, 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. However, 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—referred to as the address computation step. 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. In the data transfer step, the data is actually moved from the source address to the destination address.

The DMA transfer can occur in two different modes: burst and arbitration. In burst mode, the DMA controller transfers the entire object, which is z bytes in size, without CPU interruptions, i.e., the CPU is stalled for the whole duration of the burst if it wants to access the memory. In 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. In arbitration mode, we assume that the bus matrix employs round-robin arbitration to ensure that at least half of the system bus bandwidth is available for the CPU.

To ensure that the first DMA transfer is completed before initiating the second, we can use polling or interrupt mechanisms to ensure the channel is free. The DMA controller can generate an interrupt when a transfer is complete. For example, a `DMA_IT_TC` flag can be used to signal that the DMA transfer is done for STM32F0xx. In the case of polling, the CPU repeatedly checks a status flag in the DMA controller's status register. For example, a `DMA_FLAG_TC1` flag is used by STM32F0xx to poll the status of the DMA transfer while using the DMA channel 1. Before initializing the next DMA transfer, we use the polling mechanism in this paper to check if the previous DMA transfer is complete. Moreover, we use the polling mechanism to ensure that the DMA transfer is completed before executing the code object from SPM. More details about the DMA controller setup used during the evaluation are provided in Sect. 8.1.

We can effortlessly expand the DMA-aware multi-objective dynamic SPM allocation proposed in this paper to support DRAM and/or SRAM in addition to Flash as long as the timing and energy characterization of DRAM and/or SRAM are available.

4 Dynamic allocation problem preliminaries

In this section, we provide definitions related to dynamic allocation and some preliminary information for *DCPO* and MO_{DMA}. For convenience's sake, Table 1 provides the most frequently used symbols within this paper and their brief descriptions.

4.1 Code objects

Dynamic allocation is a compiler optimization carried out at the low-level IR stage (cf. Fig. 1). It enables the dynamic movement of objects from one memory to another during runtime. In this paper, we consider so-called code objects as dynamic allocation candidates. Before defining code objects, we first define a basic block:

Definition 1 (*Basic Block*) A basic block $b = (i_1, \ldots, i_I)$ 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 (Muchnick 1998).

Dynamic allocation of code objects requires code transformations to facilitate jumps between memories and DMA calls that copy objects from Flash to SPM during runtime. These code transformations are carried out at the low-level IR stage (cf. Fig. 1) and incur WCET and energy consumption overheads. A basic block-level dynamic allocation will need additional code to allocate every basic block, and it could inversely impact the program's WCET and energy consumption—nullifying the optimization's benefits. Therefore, the code objects should ideally be big and executed multiple times in SPM to compensate for the additional overheads. Consequently, we consider functions and loops as code object candidates.

Table 1 Symbol table

Symbol	Description
$\mathcal{C}_{(p,c)}$	c^{th} code object in the p^{th} function
C_p	Total number of code objects in the p^{th} function
$\mathtt{CO}$	Set of code objects within the program
$\mathcal{G}_g$	g^{th} global data object within the program
x	Binary decision variable vector that decides if code objects and global data objects are dynamically and statically placed in *ISPM* and *DSPM* or not, respectively
$\mathcal{A}_{(p,c,a)}$	a^{th} address object in the c^{th} code object of the p^{th} function
$A_{(p,c)}$	Total number of address objects in the c^{th} code object of the p^{th} function
$\mathbf{A}$	Number of address objects that need to be dynamically allocated according to solution vector x
$\mathcal{L}_{(p,c)}$	Set of basic blocks that define live range of the c^{th} code object of the p^{th} function
$\alpha_{(p,c,a)}$, $\beta_{(p,c,a)}$, and $\gamma_{(p,c,a)}$	Source address in Flash, size, and destination address in SPM of the address object $\mathcal{A}_{(p,c,a)}$
b_p^r	r^{th} basic block in the p^{th} function
B_p	Total number of basic blocks in the p^{th} function
$\mathbb{F}_p^r$	Execution frequency of the basic block b_p^r
$\mathbb{F}_p$	Execution frequency of the p^{th} function
α_p^r and β_p^r	Source address in Flash and size of the r^{th} basic block in the p^{th} function
$t_s(z)$	Cost of loading z 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
$\delta_{(p,c,a)}$	DMA transfer call for address object $\mathcal{A}_{(p,c,a)}$
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
$\tau_{(p,c,a)}^r$	DMA call placement cost if $\delta_{(p,c,a)}$ is placed after basic block b_p^r
$\nu_{(p,c)}^r$	Binary decision variable that is 1 if basic block b_p^r is *def* basic block of $\mathcal{C}_{(p,c)}$
$\mathbf{N}_l^j$	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
e_F	Energy consumed while reading from the Flash memory
e_S	Energy consumed while writing it to the SPM
$\mathcal{E}(x)$	Extra energy consumption cost incurred due to extra memory accesses during DMA transfers

Definition 2 (*Code Object*) A code object is a tuple $\mathcal{C}_{(p,c)} = (\mathcal{B}_{(p,c)}, \zeta)$, where $\mathcal{B}_{(p,c)}$ is a set of basic blocks associated with $\mathcal{C}_{(p,c)}$—the c^{th} code object in the p^{th} function—and ζ is a predicate that indicates whether $\mathcal{B}_{(p,c)}$ constitutes a function or a loop.

The set of basic blocks associated with a function or a loop defines the respective code object. Finally, let $\mathtt{CO} = \{\mathcal{C}_{(1,1)}, \ldots, \mathcal{C}_{(1,C_1)}, \ldots, \mathcal{C}_{(P,C_P)}\}$ be the set of code objects, where $\mathcal{C}_{(p,c)}$ is the c^{th} code object in the p^{th} function. P represents the total number of functions, and C_p is the total number of code objects in the p^{th} function.

The global data variables (or global data objects) are live for the entire program execution. Therefore, for brevity's sake, we consider them static allocation candidates and allocate them statically to *DSPM* while solving the MO_{DMA} problem. Let $\mathcal{G}_g$ be the g^{th} global data object within the program. Therefore, in this paper, the memory allocation model considers the dynamic allocation of code objects ($\mathcal{C}_{(p,c)}$) from Flash to *ISPM* during runtime and the static allocation of global data objects ($\mathcal{G}_g$) from 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 paper.

4.2 Decision variable vector

The dynamic allocation problem can be formulated as a binary decision problem, where we need to decide which code object of the program should be allocated dynamically to SPM during runtime. The decision to move a particular code object to SPM depends on the underlying minimization objectives—in this paper, we consider WCET and the energy consumption of a program as the minimization objectives.

Let $x = (x_{(1,1)}, .., x_{(1,C_1)}, .., x_{(P,C_P)}, x_{(\tilde{d}+1)}, .., x_d) \in X$ 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. $\tilde{d} = \sum_{p=1}^{P} C_p$ is the total number of code objects (functions and loops) within the program.

The subvector $x_{(1,1):(P,C_P)} \subset \{0,1\}^{\tilde{d}}$ represents the dynamic allocation decision vector for code objects, and $x_{(p,c)} \in \{0,1\}$ decides if a code object will be allocated dynamically to *ISPM* during runtime or not, i.e., if $x_{(p,c)} = 0$, the code object $\mathcal{C}_{(p,c)}$ will stay in the Flash memory, and if $x_{(p,c)} = 1$, $\mathcal{C}_{(p,c)}$ is allocated to *ISPM* during runtime. Depending on this decision subvector, the code objects are transferred dynamically to *ISPM* during runtime using the DMA controller.

For each function f_p, $x_{(p,1)}$ is the decision variable for the function itself, and the remaining decision variables, i.e., $x_{(p,2):(p,C_p)}$, are decisions for the loops within it.

The subvector $x_{(\tilde{d}+1):d} \subset \{0,1\}^{(d-\tilde{d})}$ represents the static allocation decision vector for global data variables, $(d - \tilde{d})$ is the total number of global data variables, and $x_g \in \{0,1\}\ \forall g = [(\tilde{d}+1), d]$ 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.

```
1 int a;
2 int main( void ) {
3   a = 100;
4   int b = foo( a );
5   return 0;
6 }
7 int foo( int a ) {
8   ...
9   int b = 0;
10  for( int i = 0; i <= a; ++i ) {
11    for( int j = 0; j <= i; ++j ) {
12      b++;
13    }
14    b += a;
15  }
16  return b;
17 }
```

Listing 1 Exemplary program code

Listing (1) provides a program code example with two functions (`main()` and `foo()`), two loops ($Loop_1$ and $Loop_2$) in the function `foo()`, and one global data object (`int a`). Figure 3 provides the resulting Control Flow Graph (CFG) of the exemplary program. In this example, $P = 2$, $C_1 = 1$, $C_2 = 3$, $\tilde{d} = 4$, and $d = 5$. The function `main()` consists of basic blocks `A` and `B`, and the function `foo()` consists of all remaining basic blocks from `C` to `H`. The loop code 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 code object. In `foo()`, we 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 for loops contained within the *entry point* function. 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:

$$x_{(p,1)} = 0, \text{ if } p^{\text{th}} \text{ function is the } \textit{entry point} \tag{2}$$

In Fig. 3, `main()` is the *entry point* of the program, and it is not allocated dynamically to SPM.

Moreover, if a function is allocated to SPM, then the loops within that function are also allocated to SPM. Therefore, we apply the following constraint to the decision variable vector x:

$$\forall c = [2, C_p] \quad x_{(p,c)} = 1, \text{ if } x_{(p,1)} = 1 \tag{3}$$

I.e., if the p^{th} function is placed in SPM, all the loops contained in the p^{th} function are also in SPM. In Fig. 3, if `foo()` is placed in the SPM, the loop code objects $Loop_1$ and $Loop_2$ must be placed in the SPM.

As global data objects are allocated statically to *DSPM* according to the binary decision subvector $x_{(\tilde{d}+1):d}$, the following *DSPM* size constraint is applied.

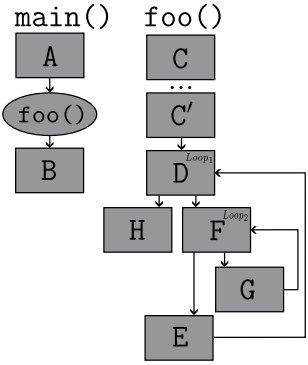


Fig. 3 CFG of the exemplary program code from Listing (1)

$$\sum_{g=(\tilde{d}+1)}^{d} S_{\mathcal{C}_g} x_g \leq S_{DSPM} \tag{4}$$

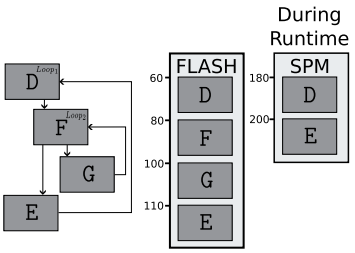
where $\forall g = [(\tilde{d}+1), d]$, $S_{\mathcal{C}_g}$ is the size of the global data object $\mathcal{G}_g$, and S_{DSPM} is the total size of the *DSPM*.

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 code 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 code object. We explain the algorithm for liveness analysis in detail in Sect. 5.2. The code objects with conflicting live ranges cannot occupy the same address ranges in the SPM. In Fig. 3, as $Loop_1$ is in use when the $Loop_2$ code object is executed, $Loop_1$ and $Loop_2$ have a liveness conflict and should not occupy the same address range in SPM.

4.3 Address objects

To dynamically allocate code objects during runtime and to determine the SPM addresses to which each code object is assigned, we solve the address assignment problem. Section 5.3 explains the algorithm to solve the address assignment problem in detail. A code 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, and then the remaining basic blocks—making it hard to assign just one SPM address to such code objects. Therefore, we use so-called address objects instead of code objects during the address assignment algorithm.

Definition 3 (*Address Objects*) An address object is defined as a set of basic blocks from the code object that are in consecutive memory addresses within the Flash, i.e.,



- Only $Loop_1$ code object is allocated dynamically to SPM during runtime.
- According to Figure (3):
  - $Loop_1$ Code Object: $\mathcal{C}_{(2,2)} = (\texttt{D}, \texttt{E})$
  - $Loop_2$ Code Object: $\mathcal{C}_{(2,3)} = (\texttt{F}, \texttt{G})$
- Address objects for $\mathcal{C}_{(2,2)}$:
  - $\mathcal{A}_{(2,2,1)} = (\texttt{D})$
  - $\mathcal{A}_{(2,2,2)} = (\texttt{E})$
- Address object for $\mathcal{C}_{(2,3)}$:
  - $\mathcal{A}_{(2,3,1)} = (\texttt{F}, \texttt{G})$

Fig. 4 Example showing difference between code objects and address objects

$$\begin{aligned}\mathcal{A}_{(p,c,a)} = \{ b_p^r \in \mathcal{C}_{(p,c)} \mid ((b_p^{(r+1)} \in \mathcal{C}_{(p,c)} \implies \alpha_p^r + \beta_p^r = \alpha_p^{(r+1)}) \vee \\ (b_p^{(r-1)} \in \mathcal{C}_{(p,c)} \implies \alpha_p^r = \alpha_p^{(r-1)} + \beta_p^{(r-1)})) \vee |\mathcal{C}_{(p,c)}| = 1 \},\end{aligned} \tag{5}$$

where $a = [1, A_{(p,c)}]$, and $A_{(p,c)}$ is the total number of address objects associated with $\mathcal{C}_{(p,c)}$—the c^{th} code 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., $\mathcal{A}_{(p,c,a)}$ is a set of b_p^r associated with $\mathcal{C}_{(p,c)}$, such that the end address of 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. If a code object consists of a single basic block ($|\mathcal{C}_{(p,c)}| = 1$), the address object is equivalent to the code object.

Figure 4 shows the nested loops from the example in Fig. 3 and their placement in the Flash. In Fig. 4, 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. If only $Loop_1$ is allocated dynamically to the SPM, we need two DMA 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 code objects and their respective memory addresses. If all basic blocks within the code objects lie in the memory consecutively, then the code object and address object are equivalent. For example, in Fig. 4, $\mathcal{C}_{(2,3)}$ 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 Fig. 4, 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 the address range that does not overlap with $\mathcal{A}_{(2,2,1)}$ and $\mathcal{A}_{(2,2,2)}$ in SPM.

In this paper, we use the DMA controller to dynamically move address objects from Flash to 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. Sect. 6). Let $\delta_{(p,c,a)}$ be the DMA transfer call for the address object $\mathcal{A}_{(p,c,a)}$. Exactly one DMA call is needed to transfer one address object.

To place the DMA call $\delta_{(p,c,a)}$ for $\mathcal{A}_{(p,c,a)}$ within the p^{th} function, *DCPO* uses a binary decision variable vector $y_{(p,c,a)} \in \{0,1\}^{(B_p-1)}$, where B_p is the total number of basic blocks in the p^{th} function. 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. Sect. 6). 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 code 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 satisfy Eqs. (2), (3), and (4) and occupy space within SPM without violating any liveness constraints. While solving the proposed MO_{DMA} problem, we generate DMA-aware dynamically allocatable code for such 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 transfer, static analysis cannot capture the execution time and energy consumption required for DMA transfers. Furthermore, due to the lack of an energy model that captures the energy consumption due to the DMA controller, we are not able to capture more accurate energy costs incurred due to arbitration and CPU stalls.

However, as mentioned in Sect. 3, we estimate the extra energy consumption costs ($\mathcal{E}(x)$) incurred due to extra memory accesses during DMA transfers in Sect. 7. Moreover, in terms of execution time, we already calculate this value ($\mathcal{T}'(y)$) by solving the *DCPO* problem in Sect. 6. 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} (cf. Sect. 7) 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 code objects during runtime.

5 Dynamic SPM allocation

In this section, we introduce liveness analysis, address assignment algorithm, and generation and analyses of dynamically allocated code at the compiler level—that enables dynamic allocation of code objects from one memory to another during runtime.

This section is divided into the following subsections: Sect. 5.1 provides the context and steps required for DMA-aware multi-objective dynamic SPM allocation and its overall picture within the compiler. Sections 5.2 and 5.3 discuss liveness analysis and address assignment algorithm, respectively, required for dynamic SPM allocation. Sections 5.4 and 5.5 present code generation and analyses of dynamically allocated code, respectively.

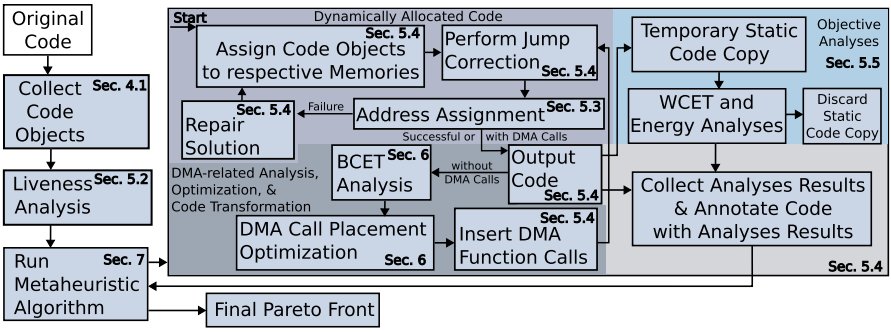


Fig. 5 DMA-aware multi-objective dynamic SPM allocation

5.1 Contextual overview

Figure 5 outlines the steps necessary to perform compiler-level DMA-aware multi-objective dynamic SPM allocation (MO_{DMA}) optimization. The steps for performing MO_{DMA} are proposed and elaborated across Sects. 5, 6, and 7, which are referenced within the figure. To generate dynamically allocatable code and solve the MO_{DMA} problem, we first identify and collect code objects from the source code. We then conduct liveness analysis to determine the live ranges of these code objects (cf. Sect. 5.2). After this, we call a metaheuristic algorithm to solve the multi-objective optimization problem proposed and discussed in detail in Sect. 7. The metaheuristic algorithm generates random binary solution vectors x that indicate which code object will be dynamically allocated to which memory. To generate dynamically allocatable code, we assign code objects to respective memories according to the solution vector x and perform appropriate code transformations (jump correction) to accommodate jumps between different memories (cf. Sect. 5.4).

We solve an address assignment problem (cf. Sect. 5.3) to find suitable memory addresses for address objects, ensuring that no liveness conflicts occur. If any address object is not assigned an SPM address, we repair the solution vector by reallocating the associated code object back to the Flash memory (cf. Sect. 5.4). Once the address assignment problem is solved, we perform a Best-Case Execution Time (BCET) analysis on the transformed code. This BCET analysis determines the BCETs for all basic blocks within the program. We use these BCET values to calculate the total execution time for the DMA transfers in arbitration mode ($\mathcal{T}'(y)$) in Sect. 6.

Next, we carry out a DMA Call Placement Optimization (*DCPO*) (proposed in Sect. 6). 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. After we insert the DMA calls, we perform jump correction and re-solve the address assignment algorithm to accommodate the changes due to DMA calls.

The optimized code containing DMA calls that dynamically copy code objects at runtime is not usable for static WCET and energy analyses. To overcome this issue and to obtain proper WCET and energy values, this optimized code is converted into

a temporary static code version by statically placing code objects in SPM. It enables us to perform static WCET and energy analyses at compile time (cf. Sect. 5.5). 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.

Algorithm 1 Function- and Loop-level Liveness Analysis

Input: Set of code objects CO
Output: Liveness data (*def*, *use*, and $\mathcal{L}$ (Live range)) for all code 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 functions
  for $c = 1 : C_p$ **do** ▷ Iterate over all code objects within the function f_p
    Initialize: $def_{(p,c)}$, $use_{(p,c)}$, and $\mathcal{L}_{(p,c)}$ as sets of basic blocks
    if $c == 1$ **then** ▷ The first code object is the function itself
      for $p' = 1 : P$ such that $p' \neq p$ **do**
        Initialize: $\mathbb{B}$ (a 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 $\mu = false$ - to check if b_c is dominated by 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
              $\mu = true$
          if $\mu == false$ **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 basic block b_d in def' **do**▷ Iterate to determine the live range of f_p
          for each basic block b_u in use' **do**
            if b_u dom b_d **then** ▷ Check if b_d dominates b_u
              $\mathcal{L}' \leftarrow \mathcal{L}' \cup \{\text{all basic blocks in between } b_d \text{ and } b_u\}$
        $def_{(p,c)} \leftarrow def'$, $use_{(p,c)} \leftarrow use'$, and $\mathcal{L}_{(p,c)} \leftarrow \mathcal{L}'$
          ▷ 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 basic block b_p in function f_p **do** ▷ Iterate over all basic blocks in f_p
        if b_p is the loop entrance basic block of the loop code object $\mathcal{C}_{(p,c)}$ **then**
          if $\mathcal{C}_{(p,c)}$ is nested loop **then**
            $def_{(p,c)} \leftarrow \{\text{entrance basic block of the outermost loop}\}$
          else ▷ Else loop's own entrance basic block is considered
            $def_{(p,c)} \leftarrow b_p$
        else if b_p is a part of the loop or its nested loops **then**
          $use_{(p,c)} \leftarrow b_p$ ▷ The rest of the basic blocks are part of the *use* set
      $\mathcal{L}_{(p,c)} \leftarrow def_{(p,c)} \cup use_{(p,c)}$
        ▷ All basic blocks in *def* and *use* define loop's live range $\mathcal{L}$

5.2 Liveness analysis

After determining code objects within the program, a liveness analysis (Appel 2004) is performed within the compiler at the function level to determine the live ranges of code objects. Algorithm (1) presents the steps for function- and loop-level liveness analysis performed within the compiler.

The algorithm takes the set of code objects (`CO`) as input and provides liveness data for all code objects as output (Lines 1 and 2). Let $\mathcal{L}_{(p,c)}$ be the set of basic blocks that define the live range of $\mathcal{C}_{(p,c)}$, the c^{th} code object in the p^{th} function. The liveness data consists of the following sets of basic blocks: *def*, *use*, and $\mathcal{L}$.

In the case of a function code object $\mathcal{C}_{(p,c)}$, *def* contains basic blocks that first call $\mathcal{C}_{(p,c)}$ from function $f_{p'}$ ($f_{p'}$ and $\mathcal{C}_{(p,c)}$ are different functions), *use* contains subsequent basic blocks from $f_{p'}$ that call $\mathcal{C}_{(p,c)}$, and $\mathcal{L}_{(p,c)}$ contains all basic blocks from $f_{p'}$ that define the live range of $\mathcal{C}_{(p,c)}$ from the *def* basic block till the last *use* basic block within every function. In Fig. 3, `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 code object $\mathcal{C}_{(p,c)}$, *def* contains either the loop entrance or the entrance basic blocks of the outermost loop, and *use* contains all basic blocks within the loop nest. All the basic blocks within the loop nest define the live range of the loop code object, i.e., $\mathcal{L}_{(p,c)}$ is a union of all basic blocks in *def* and *use*. We perform a loop nesting forest analysis (Tarjan 1973) within the compiler at the function level to get basic blocks contained within every loop and their entrance, exit, and nested loop information (Line 3).

In Fig. 3, `D` is the *def* basic block for $Loop_1$ and $Loop_2$, and `D`, `E`, `F`, and `G` define their live range. If the DMA call for transferring $Loop_2$ is placed between `D` and `F`, then the DMA call that transfers $Loop_2$ will be triggered on every iteration of $Loop_1$. Therefore, even though `F` is the loop entry of $Loop_2$, we dynamically allocate nested loop $Loop_2$ before entering $Loop_1$ to avoid iterative copying of $Loop_2$ and consider the entrance basic block of the outermost loop of the loop nest as the *def* basic block. However, it is important to note that this constraint is not strictly necessary. The *DCPO* optimization proposed in Sect. 6 can identify appropriate places for inserting DMA calls without relying on this constraint. While this constraint does help to reduce the search space of the *DCPO*, removing it will not impede the *DCPO* optimization.

We iterate over all functions and all the code objects within function f_p, initialize $def_{(p,c)}$, $use_{(p,c)}$, and $\mathcal{L}_{(p,c)}$ sets for the current code object $\mathcal{C}_{(p,c)}$, and check if $\mathcal{C}_{(p,c)}$ is a function or a loop (Lines 4-7). For functions, we iterate over all 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 $def_{(p,c)}$ or $use_{(p,c)}$ and $\mathcal{L}_{(p,c)}$ sets (Lines 8-28). def', use', and $\mathcal{L}'$ contain lists of *def*, *use*, and $\mathcal{L}$ basic blocks associated with calls for f_p in $f_{p'}$. 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 $b_{p'}$ (Line 14) is executed, then we need to consider such basic blocks as part of *def* too. Therefore, we check if the current basic block b_c that calls the

function is dominated by at least one basic block in def', and b_c is added to def' or use' accordingly (Lines 15-23). After getting def' and use' lists for $f_{p'}$, we insert all basic blocks in $\mathcal{L}'$ that define the live range of f_p in $f_{p'}$ (Lines 24-27). Finally, def', use', and $\mathcal{L}'$ are added to $def_{(p,c)}$, $use_{(p,c)}$, and $\mathcal{L}_{(p,c)}$.

For loop code objects, we iterate over all basic blocks of 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 $def_{(p,c)}$ and $use_{(p,c)}$ 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 $def_{(p,c)}$ (Lines 32-33). As the live range of a loop is defined by all basic blocks associated with the loop nest, $\mathcal{L}_{(p,c)}$ is a union of $def_{(p,c)}$ and $use_{(p,c)}$ sets (Line 38).

5.3 Address assignment algorithm

Algorithm (2) presents the steps for performing the address assignment algorithm. Algorithm (2) takes the decision vector x and the set of address objects for all $\mathcal{C}_{(p,c)}$ 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.

Algorithm 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 SPM address $\Gamma = ISPM$ start address
4: **for** $p = 1 : P$ **do** ▷ Iterate over all functions
5: **for** $c = 1 : C_p$ **do** ▷ Iterate over all code objects in the p^{th} function
6: **if** $x_{(p,c)} == 1$ **then** ▷ Check if $\mathcal{C}_{(p,c)}$ is allocated to SPM
7: **for** $a = 1 : A_{(p,c)}$ **do** ▷ Iterate over all address objects in the c^{th} code object
8: **if** $(\Gamma + \beta_{(p,c,a)}) \leq S_{ISPM}$ **then**
9: ▷ First-fit heuristic is called only if SPM has the free capacity of $\beta_{(p,c,a)}$ size
10: Call first-fit heuristic to get SPM address $\gamma_{(p,c,a)}$
11: ▷ First-fit heuristic assigns the first available SPM address to the address object
12: $\Gamma += \beta_{(p,c,a)}$ ▷ Increment Γ using the size of the current address object
13: $\epsilon += 1$ ▷ Increment ϵ to count the assigned address object
14: **else** ▷ If SPM is full, the best-fit heuristic is used to find the SPM address
15: Call best-fit heuristic to get SPM address $\gamma_{(p,c,a)}$
16: ▷ Best-fit heuristic avoids liveness conflicts by fulfilling constraint Equations (6) and (7)
17: **if** $\gamma_{(p,c,a)}$ is a valid SPM address **then**
18: $\epsilon += 1$

With the help of the source address (α) in the Flash and the total size (β) for all the address objects, we solve the address assignment problem using the first- and the

best-fit heuristics to determine their destination addresses (γ) in the SPM. We iterate over all functions and the code objects associated with the functions, and according to the solution x, if the considered code object is supposed to be allocated to SPM, we try to find the SPM address for the address objects associated with the respective code 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 8-13). 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 15-18). The best-fit heuristic tries to find the best places in the SPM for the remaining address objects.

We need to consider liveness conflicts when assigning overlapping SPM address ranges. We cannot assign two address objects associated with the same code object to the same SPM address range, i.e., for example, in Fig. 4, $\mathcal{A}_{(2,2,1)}$ 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:

$$\gamma_{(p,c,a)} + \beta_{(p,c,a)} \leq \gamma_{(p,c,a')} \vee \gamma_{(p,c,a')} + \beta_{(p,c,a')} \leq \gamma_{(p,c,a)} \quad \forall a \neq a' \tag{6}$$

Furthermore, address objects with overlapping executions (live range conflicts) are constrained to be allocated to different SPM address ranges during runtime, i.e.,

$$\begin{aligned} &\gamma_{(p,c,a)} + \beta_{(p,c,a)} \leq \gamma_{(p',c',a')} \ \vee \gamma_{(p',c',a')} + \beta_{(p',c',a')} \leq \gamma_{(p,c,a)} \\ &\quad \text{if} \quad (\mathcal{A}_{(p,c,a)} \cap (\mathcal{A}_{(p',c',a')} \cup \mathcal{L}_{(p',c')}) \neq \phi) \ \wedge ((p', c') \neq (p, c)) \end{aligned} \tag{7}$$

In this case, address objects associated with different code objects are considered, i.e., for example, in Fig. 4, if both $\mathcal{C}_{(2,2)}$ and $\mathcal{C}_{(2,3)}$ are allocated to SPM, $\mathcal{C}_{(2,2)}$ and $\mathcal{C}_{(2,3)}$ 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 Eq. (7), the best-fit heuristic utilizes the liveness data such that no two simultaneously live code objects are allocated to the same memory address range. The term $(\mathcal{A}_{(p,c,a)} \cap (\mathcal{A}_{(p',c',a')} \cup \mathcal{L}_{(p',c')})$ in Eq. (7) reflects the live range conflicts between two address objects $\mathcal{A}_{(p,c,a)}$ and $\mathcal{A}_{(p',c',a')}$. If $\mathcal{A}_{(p,c,a)}$ and the live range of $\mathcal{A}_{(p',c',a')}$ have any common basic blocks, the resulting set will be non-empty, and the constraint condition in Eq. (7) will be applied.

5.4 Code generation

Algorithm (3) presents the steps required to generate and analyze dynamically allocated code. We perform the code transformations to generate dynamically allocated code at the low-level IR stage (cf. Fig. 1). As liveness analysis (cf. Algorithm (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. Fig. 5). Algorithm (3) takes x, which decides if a code 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

code objects to SPM or the Flash depending on the decisions in x (Line 4). As we want the code to jump from the Flash to SPM and vice versa during runtime, we perform jump correction (Oehlert et al. 2016) such that previously valid jumps to address objects in the Flash are replaced by new jumps to the respective runtime address objects in SPM (Line 4). In Fig. 3, 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.

Position-Independent Code (PIC) allows code to execute without any modifications, regardless of its absolute memory address, eliminating the need for jump correction. It is commonly used in shared libraries or systems where the code needs to be loaded at different memory addresses, such as in operating systems with dynamic linking. However, PIC can increase code size due to the use of relative addressing and additional logic. For resource-constrained bare-metal real-time systems, where efficiency and predictability are critical, the overheads introduced by PIC are unnecessary. Therefore, in this paper, PIC is not used, and jump correction is applied during code generation to ensure proper execution.

Algorithm 3 Generation and Analyses 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 code objects to SPM using x and perform jump correction
5:      Perform address assignment using Algorithm (2)
6:      if (A − ϵ) ≠ 0 then
7:          Repair solution x
8:  while (A − ϵ) ≠ 0
9:  Perform BCET analysis and solve the DCPO problem
10: Insert DMA calls and perform jump correction
11: Perform address assignment after code transformations
12: Generate a static version of the code and perform analyses
```

Next, we solve the address assignment problem for the allocated code using Algorithm (2) (Line 5). As we want to eventually insert DMA transfer calls and perform some jump corrections (Line 10), it can increase the code size, and the address assignment 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 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 x by moving the code objects associated with the unassigned address objects back to the Flash, i.e., by setting their binary decision variable back

to 0 (Lines 6-7). If x is repaired, we need to generate the dynamically allocated code again by repeating steps 4 and 5 for the repaired solution (Line 8).

To copy address objects during runtime, we need DMA transfer 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 in the SPM. The values of α, β, and γ for all address objects are available after solving the address assignment problem. Address objects are copied using the DMA controller—enabling dynamic allocation from α to γ during runtime. We perform BCET analysis to calculate BCET values for all basic blocks and use them to calculate the total execution time required for DMA transfer in Sect. 6.2. Then, we solve the *DCPO* optimization (cf. Sect. 6) that provides 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).

5.5 Code analyses

We need to perform WCET and energy analyses to solve the multi-objective optimization problem. However, static analysis tools like `aiT` and `EnergyAnalyzer` cannot deal with code 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 code object at address γ behaves identical to the original code 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 code 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 code objects that should be copied dynamically onto the SPM are moved statically to the SPM, and we substitute NOP basic blocks in the Flash in place of the statically allocated code 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 code object. If the size of

the code objects allocated to SPM exceeds the actual SPM size, the static analyzer assumes a larger SPM capacity to accommodate all statically allocated code objects. 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.

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 coherency. 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 DMA controller transfers the code object to the SPM. This invalidation results in a high number of cache misses when executing the allocated code object from the SPM, worsening the WCET and energy consumption. To address this issue, the address range of the SPM could be marked as non-cacheable. However, this effectively renders the system as if no caches are present, aligning with the architecture assumed in this paper.

Finally, the jumps within the temporary code version are corrected such that the execution flow through all code objects and their static duplicates are identical to the original, dynamically allocated code. This way, all the dynamism present inherently in the final DMA-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. However, the extra costs incurred due to DMA arbitration and CPU stalls during runtime cannot be simulated by a static analyzer at compile time. Therefore, we use the DMA analyzer proposed in Sect. 6 to calculate these extra costs in terms of execution times ($\mathcal{T}'(y)$) 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.

As mentioned in Sect. 3, 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 WCET and energy analyzers to perform respective analyses of this temporary static version of the code (Algorithm (3), Line 12). Once the static version of the code is analyzed, we collect the results and discard this temporary code version.

6 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. A DMA controller can work in parallel with the CPU and carry out dynamic copying, leaving the CPU free for essential execution. The DMA controller is connected

to the CPU and memories using a bus matrix (cf. Fig. 2) that uses arbitration to grant access to either the CPU or the DMA controller, and the bus arbitration takes place when two masters need to access the same slave memory. In our case, the CPU and the DMA controller are the masters, and the Flash and the SPM are slave memories (STMicroelectronics 2022b). The DMA controller can independently transfer data from the Flash to SPM, but the CPU needs to initiate the transfer. As detailed in Sect. 3, the process for Flash-to-SPM DMA transfer involves three steps: arbitration for bus access, address computation, and data transfer.

As the Flash and SPM considered in this paper are single-ported memories, only one master can access a memory at any given time. Consequently, even if multiple DMA channels are configured for transfers, only one DMA channel can access the memory at once. To avoid unnecessary pessimism in our proposed DMA analysis model, we only initiate a single memory-to-memory DMA transfer at a time.

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 paper, we assume that if a second DMA transfer is triggered before the first one is completed, we stall the CPU to finish the first DMA transfer in burst mode. After that, the second DMA transfer will be initiated in arbitration mode.

This section is divided into the following subsections: Sect. 6.1 provides an overview of DMA call placement with examples. Section 6.2 proposes a model for the cost calculation of DMA transfer calls. Section 6.3 uses the cost calculation model and proposes the *DCPO* optimization problem.

6.1 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 code object. If the DMA transfer is incomplete, the CPU is stalled before executing the code object, and the DMA transfer takes place in burst mode. In Fig. 3, if we place DMA calls to dynamically allocate $Loop_1$ to SPM between basic blocks C′ and D, the DMA transfer of $Loop_1$ needs to end before executing D. In this case, the CPU is stalled, and the DMA transfer occurs in the burst mode, and 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 DMA Call Placement Optimization (*DCPO*), which aims to find optimal places for DMA calls within the code so that the total execution time required for DMA transfers in arbitration mode during runtime is minimized.

Figure 6 provides three exemplary cases 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. In Fig. 6a, we assume that there are a bunch of basic blocks indicated by "..." between C and K, K and L, and L and C'.

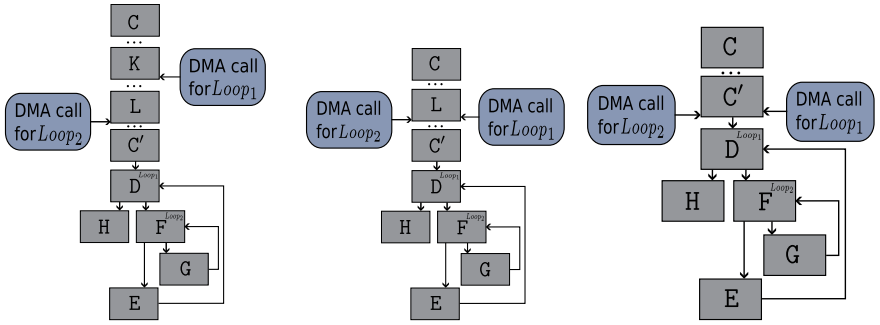


Fig. 6 Different DMA call placements

Similarly, in Figs. 6b and 6c, "..." always indicates a bunch of basic blocks between respective surrounding basic blocks.

Figure 6a shows that the DMA transfer calls are 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 transferred before initializing the DMA call for $Loop_2$, the DMA transfers of $Loop_1$ and $Loop_2$ occur with arbitration, and the CPU is granted bus access whenever needed. To ensure that the first DMA transfer is completed before initiating the second, we can use the polling mechanism explained in Sect. 3 to ensure the channel is free. For example, in Fig. 6a, if the DMA transfer of $Loop_2$ is incomplete before the execution of $Loop_1$ from SPM, the CPU is stalled until the polling flag (`DMA_FLAG_TC1`) indicates its completion.

Figure 6b shows that both DMA transfer calls are placed after basic block `L` consecutively. In this scenario, the DMA transfer of $Loop_1$ occurs in burst mode, and the CPU will be halted in the meantime, and then the DMA transfer of $Loop_2$ and the instruction execution by CPU start parallelly. CPU will be granted bus access whenever needed during DMA transfer of $Loop_2$.

Figure 6c 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: transferring of $Loop_1$ needs to be completed before initializing another DMA call, and to execute basic block `D` from SPM, $Loop_1$ 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 $Loop_2$ code 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 Figs. 6b or 6c 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 burst mode DMA transfer, then the case described in Fig. 6a 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 code objects, evoking the case described in Fig. 6c. Therefore, the proposed *DCPO* optimization considers all such cases and aims to find optimal places for all DMA transfer calls within the code, such that the total execution time required for DMA transfer in arbitration mode is minimized.

6.2 Cost calculation

In this paper, we build the DMA transfer model based on STMicroelectronics (2022b) and Whitham and Audsley (2009). We model the cost of loading z bytes from the Flash to SPM using the DMA controller in burst mode as:

$$t_s(z) = \lceil z * D_{tr} \rceil + t_\Omega, \quad \text{where} \quad t_\Omega = D_{at} + aT_F + aT_S \tag{8}$$

D_{tr} is the DMA transfer rate, 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 Flash and SPM. To perform a DMA transfer in burst mode, the bus matrix needs to grant access to the DMA controller. 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 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. $\delta_{(p,c,a)}$ is the DMA call for the a^{th} address object of the c^{th} code object in the p^{th} function.

6.2.1 Total execution time 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 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 for the DMA transfer in the burst mode for the solution x.

$$\mathcal{T}(x) = \sum_{p=1}^{P} \sum_{c=1}^{C_p} \left(\sum_{a=1}^{A_{(p,c)}} t_s(z_a) \right) x_{(p,c)} \tag{9}$$

where z_a is the size of the address object $\mathcal{A}_{(p,c,a)}$, and $t_s(z_a)$ is the cost of loading $\mathcal{A}_{(p,c,a)}$ from Flash to SPM using the DMA controller during runtime.

6.2.2 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 controller and the CPU aim to operate in parallel. However, if the CPU requires access to the bus matrix to fetch instructions from memory or to execute load or store operations, it is granted access to the bus, causing the DMA transfer to be temporarily halted. The total execution time required by the DMA controller in arbitration mode includes the times when the arbitration occurs, halting the DMA transfer. In this subsection, we will first calculate the placement cost (the extra execution time) for each basic block, assuming that the DMA transfer and the execution of the basic block occur in parallel. In the next subsection, we calculate the total execution time in the arbitration mode using the basic block-level placement cost calculated in this subsection.

Let us first define some terms used to calculate the basic block-level placement cost. $r \in [1, B_p]$ is an index to indicate a basic block (b_p^r), and B_p is the total number of basic blocks in the p^{th} function. If a DMA transfer call $\delta_{(p,c,a)}$ 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 code object $\mathcal{C}_{(p,c)}$ needs to be calculated to estimate the total execution time in the arbitration. The index j indicates the basic blocks between the r^{th} and the *def* basic block.

Let $\mathbf{N}_{\Omega}^{j} = \mathbf{N}^{j} + \mathbf{N}_{l}^{j} + \mathbf{N}_{s}^{l}$ be the maximum number of times arbitration can occur for the j^{th} basic block, i.e., the maximum number of times 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, and 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.

Let t_B^j be the BCET of the j^{th} basic block. We perform BCET analysis using a compiler-internal analyzer and calculate the BCET values for each basic block (cf. Fig. 5). 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 is added to the WCET objective of the MO_{DMA} optimization problem (cf. Eq. (24)), 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 and valid for the worst-case.

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 $\delta_{(p,c,a)}$ for the address object $\mathcal{A}_{(p,c,a)}$ is placed after the r^{th} basic block of the function f_p.

$$t_R^j = \sum_{j'=r}^{(j-1)} t_{(p,c,a)}^{j'} - t_B^j \tag{10}$$

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 ($\sum_{j'=r}^{(j-1)} t_{(p,c,a)}^{j'}$). Using t_R^j, we want to check if the DMA transfer of the address object $\mathcal{A}_{(p,c,a)}$ might end during the execution of the current j^{th} basic block. I.e., if $t_R^j \leq 0$, the DMA transfer completes during the execution of the current j^{th} basic block.

Let $t_{(p,c,a)}^j$ represent the placement cost incurred due to the j^{th} basic block between the r^{th} basic block and the *def* basic block of $\mathcal{A}_{(p,c,a)}$. Depending on the distance between the r^{th} and j^{th} basic block and the amount of bytes transferred by the DMA controller the cost value of $t_{(p,c,a)}^j$ is decided. The amount of DMA transfer done is decided using the relative placement cost t_R^j from Eq. (10). Therefore, we will apply the following equation with five different cases to calculate the cost value $t_{(p,c,a)}^j$:

$$t_{(p,c,a)}^j = \begin{cases} t_s(z_a), & \text{if } j = r, \\ (\mathbf{N}_\Omega^j + 1)t_\Omega - t_s(z_a), & \text{if } t_R^j \leq 0 \wedge j = r+1, \\ \mathbf{N}_\Omega^j t_\Omega - t_B^j, & \text{if } t_R^j > 0 \wedge j > r, \\ (\mathbf{N}_\Omega^j + 1)t_\Omega + (\sum_{j'=(r+1)}^{(j-1)} t_B^{j'} - t_s(z_a)), & \text{if } t_R^{j-1} > 0 \wedge t_R^j \leq 0 \wedge j > r+1, \\ 0, & \text{if } t_R^{j-1} \leq 0 \end{cases} \tag{11}$$

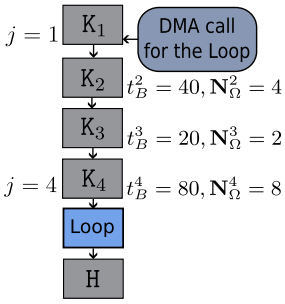
where t_Ω is the total number of cycles required to perform arbitration once (cf. Eq. (8)). We will now examine each case outlined in Eq. (11) and clarify them using examples in Fig. 7. Figure 7 presents an exemplary CFG with one DMA transfer call that transfers a "Loop" and three example scenarios with different DMA call costs in burst mode ($t_s(z_a)$) to show the calculations for DMA call placement cost. In Fig. 7a, the DMA transfer call is placed between basic blocks $\mathtt{K}_1$ and $\mathtt{K}_2$, i.e., the DMA transfer call is placed after $r = 1$ basic block. We assume the BCET values (t_B^j) and $\mathbf{N}_\Omega^j$ values for $\mathtt{K}_2$, $\mathtt{K}_3$, and $\mathtt{K}_4$ basic blocks to perform representative calculations in illustrations Figs. 7c, 7d, and 7b. Depending on the value of j and t_R^j, the corresponding case in Eq. (11) decides the cost incurred by the j^{th} basic block as follows:

Case 1 $t_{(p,c,a)}^j = t_s(z_a)$, if $j = r$ **:**

As the DMA call $\delta_{(p,c,a)}$ is placed after the r^{th} basic block, $j = r$ is the first basic block where we start counting the cost for $t_{(p,c,a)}^j$. The placement cost for the $(j = r)^{\text{th}}$ basic block is equal to the cost of transferring the address object $\mathcal{A}_{(p,c,a)}$ of size z_a bytes in burst mode ($t_s(z_a)$). If any other DMA call ($\delta_{(p',c',a')}$) for some $\mathcal{A}_{(p',c',a')}$ is placed immediately after $\delta_{(p,c,a)}$, the DMA

transfer for $\mathcal{A}_{(p,c,a)}$ is carried out in the burst mode before initiating the second DMA call $\delta_{(p',c',a')}$ for $\mathcal{A}_{(p',c',a')}$ (cf. Sect. 3 and Fig. 6b). In this case, the $j = r$ case accounts for the total cost of transferring $\mathcal{A}_{(p,c,a)}$ in the burst mode. However, if the DMA transfer of $\mathcal{A}_{(p,c,a)}$ 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 $t^j_{(p,c,a)}$ is calculated for $j > r$ cases using the other four cases in Eq. (11). In Figs. 7c, 7d, and 7b, $t^1_{(p,c,a)}$ is always equal to the respective $t_s(z_a)$ value.

Case 2 $t^j_{(p,c,a)} = (\mathbf{N}^j_\Omega + 1)t_\Omega - t_s(z_a)$, if $t^j_R \leq 0 \wedge j = r + 1$ **:**



(a) Exemplary CFG with DMA call

- $t^1_{(p,c,a)} = t_s(z_a) = 40$
- $t^2_R = 0 \implies t^2_{(p,c,a)} = (\mathbf{N}^2_\Omega + 1) * t_\Omega - t_s(z_a) = -30$
- $t^3_R = -10$ and $t^2_R \leq 0 \implies t^3_{(p,c,a)} = 0$
- $t^4_R = -70$ and $t^3_R \leq 0 \implies t^4_{(p,c,a)} = 0$
- Total cost $= \sum_{j=1}^{4} t^j_{(p,c,a)} = 10$ cycles

(b) Burst mode DMA call cost $t_s(z_a) = 40$ cycles

- $t^1_{(p,c,a)} = t_s(z_a) = 120$
- $t^2_R = 80 \implies t^2_{(p,c,a)} = \mathbf{N}^2_\Omega t_\Omega - t^2_B = -32$
- $t^3_R = 68 \implies t^3_{(p,c,a)} = \mathbf{N}^3_\Omega t_\Omega - t^3_B = -16$
- $t^4_R = -8 \implies t^4_{(p,c,a)} = (\mathbf{N}^4_\Omega + 1)t_\Omega + (\sum_{j'=2}^{3} t^{j'}_B - t_s(z_a)) = -42$
- Total cost $= \sum_{j=1}^{4} t^j_{(p,c,a)} = 30$ cycles

(c) Burst mode DMA call cost $t_s(z_a) = 120$ cycles

- $t^1_{(p,c,a)} = t_s(z_a) = 200$
- $t^2_R = 160 \implies t^2_{(p,c,a)} = \mathbf{N}^2_\Omega t_\Omega - t^2_B = -32$
- $t^3_R = 148 \implies t^3_{(p,c,a)} = \mathbf{N}^3_\Omega t_\Omega - t^3_B = -16$
- $t^4_R = 152 \implies t^4_{(p,c,a)} = \mathbf{N}^4_\Omega t_\Omega - t^4_B = -64$
- Total cost $= \sum_{j=1}^{4} t^j_{(p,c,a)} = 88$ cycles

(d) Burst mode DMA call cost $t_s(z_a) = 200$ cycles

Fig. 7 Exemplary DMA placement cost calculation: We assume that $j \in [1, 4]$, $r = 1$, and arbitration time $t_\Omega = 2$ cycles

The relative placement cost (t_R^j) indicates if the DMA transfer is completed during the execution of the current basic block ($t_R^j \leq 0$). If the DMA transfer of $\mathcal{A}_{(p,c,a)}$ is completed during the execution of the $(j = r + 1)^{\text{th}}$ basic block, then the total cost of transferring $\mathcal{A}_{(p,c,a)}$ should only include the arbitration costs that were incurred during the execution of the $(j = r + 1)^{\text{th}}$ basic block, i.e., $(\mathbf{N}_\Omega^j + 1)t_\Omega$. We add a "+1" to compensate for the arbitration cost t_Ω from $t_s(z_a)$ that is being subtracted.

In Fig. 7b, we assume the DMA call cost to transfer the Loop in burst mode is $t_s(z_a) = 40$ cycles. From calculations, we can see that $(t_R^2 = -30) \leq 0$ and $(j = 2) = (r + 1)$ as $r = 1$. Therefore, the second case in Eq. (11) is already satisfied at the basic block $\mathtt{K}_2$. Consequently, the total cost incurred to transfer the Loop is 10 cycles, which is equal to the arbitration cost $((\mathbf{N}_\Omega^j + 1)t_\Omega)$ incurred until the execution of $\mathtt{K}_2$. As the DMA transfer of the Loop is over during the execution of $\mathtt{K}_2$ itself, the value of t_R^3 and t_R^4 is set to zero using the fifth case in Eq. (11), which is explained later in detail.

Case 3 $t^j_{(p,c,a)} = \mathbf{N}^j_\Omega t_\Omega - t^j_B$, if $t^j_R > 0 \wedge j > r$ **:**

If the DMA transfer of $\mathcal{A}_{(p,c,a)}$ is not completed ($t_R^j > 0$) during the execution of the j^{th} basic block, where $j > r$, then the third case in Eq. (11) calculates the number of cycles during which the DMA and the CPU operates 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 $t^j_{(p,c,a)}$ cost is pessimistic and remains valid for the worst-case scenario.

In Fig. 7c, we assume the DMA cost to transfer the Loop in burst mode is $t_s(z_a) = 120$ 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 Eq. (11) is satisfied for both t_R^2 and t_R^3. The negative values of $t^2_{(p,c,a)}$ and $t^3_{(p,c,a)}$ indicate that the DMA transfer of the Loop and the execution of basic blocks $\mathtt{K}_2$ and $\mathtt{K}_3$ are happening in parallel.

In Fig. 7d, we assume the cost to transfer the Loop in burst mode is $t_s(z_a) = 200$ cycles. Similar to the case in Fig. 7c, both t_R^2 and t_R^3 are greater than zero and satisfy the third case in Eq. (11). However, we can also see from calculations that $(t_R^4 = 152) > 0$ and $(j = 4) > (r = 1)$, i.e., the third case in Eq. (11) is also satisfied for the last basic block $\mathtt{K}_4$ before the *def* basic block of the Loop. Consequently, the DMA transfer is not completed, and the CPU is stalled to complete the DMA transfer 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 DMA transfer in burst mode.

Case 4 $t^{j}_{(p,c,a)} = (\mathbf{N}^{j}_{\Omega} + 1)t_{\Omega} + (\sum_{j'=(r+1)}^{(j-1)} t^{j'}_{B} - t_s(z_a)),$ if $t^{j-1}_{R} > 0 \wedge t^{j}_{R} \leq 0 \wedge j > r+1$ **:**

If the DMA transfer of $\mathcal{A}_{(p,c,a)}$ is completed during the execution of any basic block where $j > r+1$, the fourth case in Eq. (11) is satisfied. The condition $t^{j-1}_{R} > 0$ indicates that the DMA transfer was not completed during the execution of the $(j-1)^{\text{th}}$ basic block, while the condition $t^{j}_{R} \leq 0$ confirms that the DMA transfer is now complete. When the fourth case in Eq. (11) 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 $t_s(z_a)$ that is being subtracted. Additionally, the $\sum_{j'=(r+1)}^{(j-1)} t^{j'}_{B}$ term is added to compensate for all the BCET values that were being subtracted from all previous j' $^{\text{th}}$ basic blocks. Consequently, if the DMA transfer is not interrupted by another DMA transfer call or if it is completed before the execution of $\mathcal{A}_{(p,c,a)}$ (as illustrated in Fig. 6a), the total number of cycles required for the DMA transfer in arbitration mode will include of all possible arbitration costs.

In Fig. 7c, we assume the DMA cost to transfer the loop in burst mode is $t_s(z_a) = 120$ cycles. From calculations, we can see that $(t^{4}_{R} = -8) \leq 0$ and $(j = 4) > (r+1)$ as $r = 1$. Therefore, the fourth case in Eq. (11) is satisfied. The loop transfer will finish during the execution of $\mathtt{K}_4$, 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 $\mathtt{K}_4$.

Case 5 $t^{j}_{(p,c,a)} = 0,$ if $t^{j-1}_{R} \leq 0$ **:**

Lastly, if the DMA transfer of $\mathcal{A}_{(p,c,a)}$ is complete before the execution of the j^{th} basic block, i.e., $t^{j-1}_{R} \leq 0$, then the $t^{j}_{(p,c,a)}$ cost for all the remaining basic blocks is set to zero.

In Fig. 7b, the DMA transfer of the Loop is over during the execution of $\mathtt{K}_2$ itself, and the DMA costs for $\mathtt{K}_3$ and $\mathtt{K}_4$ are set to zero, i.e., $t^{3}_{(p,c,a)} = t^{4}_{(p,c,a)} = 0$, as this fifth case in Eq. (11) is satisfied, i.e., for $j = 3$, $t^{j-1}_{R} = t^{2}_{R} = -30 \leq 0$, and for $j = 4$, $t^{j-1}_{R} = t^{3}_{R} = -10 \leq 0$.

Using the examples in Fig. 7, we can see possible scenarios where each case in Eq. (11) is invoked. In the next subsection, we use Eq. (11) to calculate the total execution time in arbitration mode to perform DMA transfers during runtime.

6.2.3 Total execution time in arbitration mode

To begin, let's define some terms that will be used to calculate the total execution time in arbitration mode. As mentioned in Sect. 4.3, $y^r_{(p,c,a)} \in \{0,1\}\,\forall r = [1,(B_p-1)]$ is the binary decision variable vector that decides whether the DMA transfer call $\delta_{(p,c,a)}$ will be placed after the r^{th} basic block (b^r_p) within the p^{th} function or not, i.e.,

$$y^r_{(p,c,a)} = \begin{cases} 1, & \text{if } x_{(p,c)} = 1 \wedge \delta_{(p,c,a)} \text{ is placed after } b^r_p, \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

As previously mentioned, B_p is the total number of basic blocks in the p^{th} function. 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 $y_{(p,c,a)}$ for $\delta_{(p,c,a)}$ is $(B_p - 1)$. Moreover, let $v_{(p,c)} = (v^1_{(p,c)}, .., v^{(B_p-1)}_{(p,c)})$ represent a binary vector for the code object $\mathcal{C}_{(p,c)}$, where

$$v^q_{(p,c)} = \begin{cases} 1, & \text{if } b^q_p \text{ is the } \textit{def} \text{ basic block of } \mathcal{C}_{(p,c)}, \\ 0, & \text{otherwise} \end{cases} \tag{13}$$

As *def* basic blocks of code objects are always the same, $v_{(p,c)}$ is a constant binary vector in the *DCPO* optimization. Additionally, $v^q_{(p,c)} = 1$ implies that the q^{th} basic block is the *def* basic block of the code object $\mathcal{C}_{(p,c)}$.

$$\eta^j_{(p,c,a)} = \begin{cases} 1, \text{ if } j = r \vee (\sum_{c=1}^{C_p} \sum_{a'=1, a' \neq a}^{A_{(p,c)}} y^j_{(p,c,a')} = 0 \wedge \\ \qquad\qquad \eta^k_{(p,c,a)} = 1\, \forall k \in j : k < j), \\ 0, \text{ otherwise} \end{cases} \tag{14}$$

Next, $\eta^j_{(p,c,a)}$ is a binary constant that indicates if the basic block-level placement cost $t^j_{(p,c,a)}$ (cf. Eq. (11)) for the j^{th} basic block will be considered or not while calculating the total placement cost. As we can see from Eq. (14), $\eta^j_{(p,c,a)}$ is always '1' for the r^{th} basic block, i.e., $j = r$. Even if any other DMA call $\delta_{(p',c',a')}$ is placed immediately after the current DMA call $\delta_{(p,c,a)}$, $\eta^j_{(p,c,a)} = 1$ for $j = r$ ensures that at least the cost to transfer DMA call $\delta_{(p,c,a)}$ in burst mode is incurred. This case works similarly to the first case in Eq. (11). Moreover, Fig. 6b 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, $\eta^j_{(p,c,a)}$ 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 the rest of the basic blocks, $\eta^j_{(p,c,a)}$ is '1' only if no other DMA transfer calls are placed until $(j = (q-1))^{\text{th}}$ basic block. As previously mentioned, q^{th} basic block is the *def* basic block of the code object $\mathcal{C}_{(p,c)}$ and all the address objects $\mathcal{A}_{(p,c,a)}$ associated with $\mathcal{C}_{(p,c)}$. If another DMA call $\delta_{(p,c,a')}$ is placed at the k^{th} basic block that is before the $(q-1)^{\text{th}}$ basic block, then $\eta^j_{(p,c,a)}$ is set to '0' from $j = (k+1)$ till $j = (q-1)$. Figure 6a addresses this scenario with an example, where $\eta^j_{(p,c,a)}$, 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 $\eta^j_{(p,c,a)}$ is '1' for $Loop_2$ for all basic blocks.

Moreover, in the examples in Fig. 7, we assumed $\forall j \in [1,4]\, \eta^j_{(p,c,a)} = 1$. However, if another DMA call interrupts the first DMA call, then $\eta^j_{(p,c,a)} = 0 \;\; \forall j > k$ (cf. Eq. (14)), and the cost calculations for the DMA transfer are handled similarly to calculations shown in the example in Fig. 7d.

In Sect. 6.2.2, we calculated the basic block-level placement cost ($t^j_{(p,c,a)}$) for each j^{th} basic block in arbitration mode between the r^{th} and the q^{th} basic block. The DMA transfer call $\delta_{(p,c,a)}$ is placed after the r^{th} basic block, and the q^{th} basic block is the *def* basic block of $\mathcal{C}_{(p,c)}$ and corresponding $\mathcal{A}_{(p,c,a)}$. Using $t^j_{(p,c,a)}$, we can determine the overall DMA call placement cost in arbitration mode incurred when $\delta_{(p,c,a)}$ is placed at the r^{th} basic block in the p^{th} function. Let $\tau^r_{(p,c,a)}$ represent the DMA call placement cost in arbitration mode.

$$\tau^r_{(p,c,a)} = \begin{cases} \sum_{j=r}^{(q-1)} t^j_{(p,c,a)} \eta^j_{(p,c,a)}, & \text{if } \nu^q_{(p,c)} = 1 \;\wedge\; r < (q-1) \\ t_s(z_a), & \text{if } \nu^q_{(p,c)} = 1 \;\wedge\; r = (q-1) \\ 0, & \text{otherwise} \end{cases} \tag{15}$$

The first case in Eq. (15) captures the following cases: the DMA transfer with the arbitration (Fig. 6a) and if other DMA calls are placed after the current DMA call $\delta_{(p,c,a)}$ (Fig. 6b). The second case in Eq. (15) captures cases like Fig. 6c, where the DMA call is placed immediately before the code object, and the DMA transfer occurs in the burst mode.

Case 1 $\tau^r_{(p,c,a)} = \sum_{j=r}^{(q-1)} t^j_{(p,c,a)} \eta^j_{(p,c,a)}$, if $\nu^q_{(p,c)} = 1 \;\wedge\; r < (q-1)$**:**

As previously mentioned, $\nu^q_{(p,c)} = 1$ 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 code object, i.e., $r < (q-1)$ (cf. Figs. 6a and 6b). In this case, we use the basic block-level placement cost ($t^j_{(p,c,a)}$) from Eq. (11) to calculate the DMA call placement cost incurred by $\delta_{(p,c,a)}$. When the DMA call is placed at $(j = r)^{\text{th}}$ basic block such that $r < (q-1)$, we perform a summation of $t^j_{(p,c,a)}$ incurred due to each basic block from $j = r$ until $j = (q-1)$. Depending on the value of $\eta^j_{(p,c,a)}$, the $t^j_{(p,c,a)}$ cost is either added to $\tau^r_{(p,c,a)}$ or not. Once the value of $\eta^j_{(p,c,a)}$ becomes zero, $\eta^j_{(p,c,a)}$ is zero for the rest of the basic blocks.

Case 2 $\tau^r_{(p,c,a)} = t_s(z_a)$, if $\nu^q_{(p,c)} = 1 \;\wedge\; r = (q-1)$**:**

If the DMA call $\delta_{(p,c,a)}$ is placed immediately before the *def* basic block of code object $\mathcal{C}_{(p,c)}$, i.e., $r = (q-1)$, the CPU is stalled. Then, the DMA controller transfers the address object $\mathcal{A}_{(p,c,a)}$ of the corresponding $\mathcal{C}_{(p,c)}$ in burst mode—the resulting cost is $t_s(z_a)$. The code object is executed from the SPM once the DMA transfer is complete.

Case 3 $\tau^r_{(p,c,a)} = 0$, otherwise **:**

Since the DMA transfer must be completed before the execution of the code object can begin, the value of the cost $\tau^r_{(p,c,a)}$ is set to zero for all the basic blocks where $r > (q-1)$.

Finally, let $\mathcal{T}'(y)$ represent the total execution time 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.

$$\mathcal{T}'(y) = \sum_{p=1}^{P} \sum_{c=1}^{C_p} \left(\sum_{a=1}^{A_{(p,c)}} \sum_{r=1}^{B_p-1} \mathbb{F}^r_p \tau^r_{(p,c,a)} y^r_{(p,c,a)} \right) x_{(p,c)} \tag{16}$$

The binary decision variable vector $y_{(p,c,a)}$ decides exactly where the DMA transfer call $\delta_{(p,c,a)}$ will be placed within the function. If $\delta_{(p,c,a)}$ is placed at the r^{th} basic block within the p^{th} function, then the cost $\tau^r_{(p,c,a)}$ calculated for the r^{th} basic block is added to the total $\mathcal{T}'(y)$.

$x_{(p,c)}$ is the decision vector that decides whether a code object should be dynamically allocated to SPM or not. As the *DCPO* optimization problem is solved after performing address assignment and BCET analysis, the decision vector $x_{(p,c)}$ is already set for the particular instance of *DCPO* evaluation (cf. Fig. 5). Therefore, the $x_{(p,c)}$ decision vector is treated as a constant vector in Eq. (16).

6.3 *DCPO* optimization problem

We want to find optimal places within the program to place the DMA calls such that the total execution time $\mathcal{T}'(y)$ 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 code object 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 f_1 is being called for the first time from f_2. As we do not place the DMA calls for the corresponding code object after its *def* basic block, we can constrain those basic

blocks. $\nu^{q}_{(p,c)} = 1$ implies that the q^{th} basic block is the *def* basic block of the address object $\mathcal{A}_{(p,c,a)}$.

$$y^{r}_{(p,c,a)} = 0 \quad \forall r \geq q \mid \nu^{q}_{(p,c)} = 1 \tag{17}$$

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. $\texttt{isPartOfLoop}(b^{r}_{p})$ is a Boolean function that checks if the basic block b^{r}_{p} is part of a loop or not. Although this constraint is applied to the *DCPO*, it is not strictly necessary. As mentioned in Sect. 5.2, the *DCPO* can still provide valid solutions without this constraint. However, applying it helps reduce the search space for *DCPO* optimization.

$$y^{r}_{(p,c,a)} = 0 \, \forall r \mid \texttt{isPartOfLoop}(b^{r}_{p}) = \mathit{true} \tag{18}$$

3. DMA calls are placed only after the basic blocks that dominate the code objects. If we place a basic block within a branch of an if-else statement, which, when executed, never leads to the execution of the code object, then triggering the DMA transfer within such a branch of the if-else statement is useless. Therefore, if the code 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 code object, i.e., $\nu^{q}_{(p,c)} = 1$.

$$y^{r}_{(p,c,a)} = 0 \, \forall r \mid \nu^{q}_{(p,c)} = 1 \wedge \neg(b^{r}_{p} \text{ dom } b^{q}_{p}) \tag{19}$$

4. If two address objects belonging to different code objects—$\mathcal{A}_{(p,c,a)}$ and $\mathcal{A}_{(p',c',a')}$, where $c \neq c'$—share an address range within SPM, and if $\mathcal{A}_{(p,c,a)}$'s *def* basic block is executed after $\mathcal{A}_{(p',c',a')}$, we must place the DMA call for $\mathcal{A}_{(p,c,a)}$ after $\mathcal{A}_{(p',c',a')}$'s execution to avoid overwriting $\mathcal{A}_{(p',c',a')}$ from SPM before its execution is finished. q_{exit} is the index for the exit basic block of the address object $\mathcal{A}_{(p',c',a')}$, and q_1 and q_2 are the indices for the *def* basic blocks for $\mathcal{A}_{(p,c,a)}$ and $\mathcal{A}_{(p',c',a')}$, respectively.

$$\begin{aligned} y^{r}_{(p,c,a)} = 0 \, \forall \, r \leq q_{exit} \mid c \neq c' \wedge \\ (\, q_1 > q_2 \mid \nu^{q_1}_{(p,c)} = 1 \ \wedge \ \nu^{q_2}_{(p',c')} = 1 \,) \wedge \\ \gamma_{(p',c',a')} \leq \gamma_{(p,c,a)} + \beta_{(p,c,a)} \wedge \\ \gamma_{(p,c,a)} \leq \gamma_{(p',c',a')} + \beta_{(p',c',a')} \end{aligned} \tag{20}$$

5. Exactly one DMA call is placed for each address object.

$$\sum_{r=1}^{B_p} y^{r}_{(p,c,a)} = 1 \tag{21}$$

6. The total execution time with arbitration always remains less than or equal to the total burst mode execution time.

$$\mathcal{T}'(y) \leq \mathcal{T}(x) \tag{22}$$

The final minimization problem for the DMA call placement optimization is mathematically formulated as follows:

$$\begin{aligned} \min_{y} \quad & \mathcal{T}'(y) \\ \text{subject to} \quad & \text{Eq. (17), Eq. (18), Eq. (19),} \\ & \text{Eq. (20), Eq. (21), and Eq. (22)} \end{aligned} \tag{23}$$

DCPO is performed after the first address assignment stage in Algorithm (3). 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 (cf. Fig. 5).

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 any constraint described in this section, we try to repair the individual by flipping the bits and checking the constraint condition. In the worst case, *DCPO* will place all the DMA calls before the *def* basic blocks of the code objects (cf. Fig. 6c), where the DMA call placement cost is maximum, but all the constraint conditions are satisfied. More details regarding the metaheuristic algorithm run are presented in Algorithm (4) and Sect. 7 in the context of MO_{DMA}.

To solve *DCPO* using ILPs, 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 a constant vector while solving DCPO. Therefore, they can be implemented easily as an ILP model within the compiler. We solve the ILPs by feeding the ILP model as input to an ILP solver called `Gurobi` (Gurobi Optimization Inc 2024) within the compiler.

By solving *DCPO*, we want to find the optimal decision vector y that satisfies all constraints described in this section and takes minimum extra time to perform required DMA transfers. During evaluations, we compare the quality of the obtained solutions to determine the best approach—ILP, FPA, or SPEA—to solve the *DCPO* problem.

Additionally, since we currently lack an energy model for a DMA controller, we do not consider energy consumption as 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 in arbitration mode 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 paper.

7 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 WCET and energy consumption of the program. Figure 5 provides an overview of all the steps involved in performing MO_{DMA}. 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. The optimality of multi-objective optimizations is not straightforward. Therefore, the following concepts of dominance and Pareto optimality are generally used to evaluate the quality of solutions for such optimization problems (Zitzler 1999; Emmerich and Deutz 2018; Ehrgott 2005).

Definition 4 (*Pareto Dominance*) For a minimization problem, let $x, y \in X$ be any two decision vectors, and $F_i(x)$ and $F_i(y)$ are objective values of x and y, where $i = [1, N]$ and N is the total number of objectives. Then, $x \preceq y$, i.e., x dominates y, if $F_i(x) \leq F_i(y)$ for all objectives, and there is at least one objective $F_i(.)$ where $F_i(x) < F_i(y)$ holds. For a maximization problem, the inverse inequations will be true.

Definition 5 (*Pareto Optimal Solution*) A decision variable vector $x \in X$ that is not dominated by any other solution is called Pareto optimal solution.

Definition 6 (*Pareto Optimal Set and Pareto Front*) The Pareto optimal set is the set of all Pareto optimal solutions, and the corresponding objective vectors form the Pareto front.

As a direct "greater than" or "less than" relation cannot be established between two objective vectors in multi-objective optimization, the metaheuristic algorithms use these Pareto optimality concepts to determine nondominated solution sets during optimization. Knowles et al. (2006) 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, any multi-objective optimization aims to identify an approximated Pareto front that closely resembles the true Pareto front.

The multi-objective optimization problem considered in this paper is quite complex, with no known function that can formulate the objective functions (WCET and energy consumption) of this problem. So, we need to evaluate the objectives analytically using WCET and energy analyzers for every possible

version of the source code. Moreover, the number of decision variables can vary depending on the source code, leading to a large search space. Exhaustively exploring the entire search space can be computationally infeasible. Identifying the entire Pareto front, which represents the trade-offs between WCET and energy objectives across the complete search space, can be an extremely complex task. Therefore, finding the true Pareto front for such a problem is ambitious and not feasible in most cases. To address these challenges, we use metaheuristic algorithms—FPA and SPEA—to explore the search space and converge towards a set of solutions that approximate the Pareto front (Knowles et al. 2006). For brevity's sake, when we talk about the Pareto front in this paper, we are referencing such an approximation of the Pareto front.

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

$$\begin{aligned} \min_{x} \quad & F(x) = ((F_1(x) + \mathcal{T}'(y)), (F_2(x)) + \mathcal{E}(x)) \\ \text{subject to} \quad & \text{Eq. (2), Eq. (3), Eq. (4), and } (\mathbf{A} - \epsilon) = 0 \end{aligned} \tag{24}$$

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 WCET and energy consumption of a program are calculated using the industry-standard static analyzer tools called `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 Eq. (1). However, we can access the WCET and energy information at the granularity of a basic block within the compiler. More details about `aiT` and `EnergyAnalyzer` are mentioned in Sect. 8.1.

$$\mathcal{E}(x) = \sum_{p=1}^{P} \sum_{c=1}^{C_p} \left(\mathbb{F}_p \sum_{a=1}^{A_{(p,c)}} (e_F + e_S) \beta_{(p,c,a)} \right) x_{(p,c)} \tag{25}$$

$\mathcal{E}(x)$ 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, and e_S is the energy cost for writing to the SPM. From Eq. (1), we set the values of e_F and e_S as: $e_F = 1.037625441$ nJ and $e_S = 1.031341343$ nJ. $\beta_{(p,c,a)}$ is the size of the address object that is being dynamically allocated to the SPM, and $\mathbb{F}_p$ is the execution frequency of the p^{th} function. As DMA transfer calls are not placed within a loop of a function (cf. Eq. (18)), we multiply the energy cost incurred due to extra memory accesses during DMA transfers only with the execution frequency of the function.

The value of $\mathcal{T}'(y)$ is obtained from the *DCPO* solved in Sect. 6. Equations (2) and (3) are constraints on the solution subvector $x_{(1,1):(P,C_P)}$ for the dynamic

allocation of code objects. Equation (4) is a constraint on the solution subvector $x_{(\tilde{d}+1):d}$ for the static allocation of global data objects. The fourth constraint $(\mathbf{A} - \epsilon) = 0$ says that the address assignment algorithm should return 0 for the solution x.

Algorithm 4 DMA-aware Multi-Objective Dynamic SPM Allocation (MO_{DMA})

```
1: Recognize and collect code objects
2: Perform Liveness Analysis using Algorithm (1)
3: for i = 1 : I do                ▷ Run metaheuristic algorithm for I different initial populations
4:     Randomly initialize the initial population of size N
5:     for n = 1 : N do
6:         Call Algorithm (3)
7:     while #generations ≤ maxGen do
8:         Update Individuals using respective update operators
9:         for Each updated Individual do
10:            Call Algorithm (3)
11:        Update the next generation using the selection operator
12:    Collect the Pareto optimal solution set
13: return Final Pareto optimal solution set
```

Algorithm (4) presents the compiler-level MO_{DMA} optimization. To initialize the algorithm, we recognize and collect all the code objects by performing standard control flow and depth-first analyses within the compiler (Algorithm (4), Line 1). Then, we perform the liveness analysis to determine the live ranges of all code objects (Algorithm (4), Line 2). We use the metaheuristic algorithms (FPA and SPEA) to solve the MO_{DMA} problem. We perform evaluations using I different sets of initial populations and run the metaheuristic algorithm to solve the MO_{DMA} problem I times for different initial populations (Algorithm (4), Line 3). The initial population of metaheuristic algorithms is initialized randomly (Algorithm (4), 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 I metaheuristic algorithm runs to reduce the influence of randomness on the final results (Algorithm (4), Lines 12 and 13).

Algorithm (3) is called to generate the dynamically allocated code for all the individuals in the initial population (Algorithm (4), Lines 5 and 6). After Algorithm (3) 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 set as the stopping criterion, as we want the metaheuristic algorithm to terminate at some point

(Algorithm (4), Line 7). The metaheuristic algorithm updates the individuals at every generation (Algorithm (4), Line 8). FPA uses local and global pollination operators (Yang 2012), and SPEA uses recombination and mutation operators to update each individual at every iteration (Zitzler 1999) to explore the search space. Algorithm (3) is called to generate and analyze the dynamically allocated code for updated individuals, and their objective values are collected (Algorithm (4), 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 (Emmerich and Deutz 2018) and pass them to the next iteration (Algorithm (4), Line 11). After the stopping criterion is met, the algorithms output the Pareto front (Algorithm (4), Line 12). We collect the Pareto fronts for each initial population and generate the final Pareto optimal solution set using Pareto dominance (Algorithm (4), Line 13).

8 Evaluations

In this section, we compare the proposed MO_{DMA} approach with MO_{MEM}, SO_D, SO_{DMA}, MO_S, and the *BaseLine* evaluation (cf. Table 2). We solve all the multi-objective optimizations using two metaheuristic algorithms—FPA and SPEA—, and all the single-objective optimizations are solved using ILPs. Moreover, the DMA call placement optimization (*DCPO*), proposed in Sect. 6, is solved using all three approaches—FPA, SPEA, and ILPs.

In Sect. 8.1, we describe the overall evaluation setup used during these evaluations. In Sect. 8.2, we solve the single-objective *DCPO* problem proposed in this paper using FPA, SPEA, and ILP methods and evaluate the quality of solutions obtained using each method. In Sects. 8.3 and 8.4, 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 Sect. 8.3, we solve the proposed MO_{DMA} optimization using *DCPO*–FPA, and in Sect. 8.4, we use *DCPO*–ILP while evaluating MO_{DMA}. Furthermore, in Sect. 8.5, 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}.

8.1 Evaluation setup

We implemented all the approaches within the WCET-aware C Compiler (WCC) (cf. Sect. 3 and Fig. 1). All optimizations are solved on an Intel XEON Gold 6146 server, with 48 cores clocked at 3.2 GHz with 1.48TiB RAM. The evaluations are conducted on the ARM Cortex-M0 microcontroller's fork of the WCC compiler. To

Table 2 Symbol table for the optimizations performed during evaluation

Symbol	Description
MO_{DMA}	DMA-aware multi-objective dynamic SPM allocation proposed in this paper
DCPO	Single-objective DMA call placement optimization proposed in this paper
MO_{MEM}	Dynamic SPM allocation-based multi-objective optimization (Jadhav and Falk 2023)
SO_D	ILP-based single-objective dynamic SPM overlay optimization (Verma and Marwedel 2007)
SO_{DMA}	Function-level DMA-aware single-objective ILP-based dynamic SPM allocation (Kim et al. 2017)
MO_S	Multi-objective static SPM allocation (Jadhav and Falk 2019, 2022)
BaseLine	The complete program is executed from the Flash memory

solve ILP-based optimization problems, an ILP solver called `Gurobi` (Gurobi Optimization Inc 2024) is integrated within the WCC compiler.

WCET and energy analyses are conducted using the industry-standard analyzers called `aiT` (AbsInt Angewandte Informatik, GmbH 2024) and `EnergyAnalyzer` (TeamPlay Consortium 2020), respectively. The WCET and energy analysis occurs after the low-level IR stage within the WCC compiler (cf. Fig. 1). The ARM Cortex-M0's energy model used within `EnergyAnalyzer` was developed using the STM32F0-Discovery board (Wegener et al. 2023). Therefore, we employ the same STM32F0xx microcontroller (STMicroelectronics 2022a) implementation within the WCC compiler. Wegener et al. (2023) and Nikov et al. (2022) proposed an energy model with its associated costs that are used within the `EnergyAnalyzer` during energy analysis (cf. Sect. 3). The values for e_F and e_S in Eq. (25) 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 (ST Microelectronics 2022a). The access latencies considered are 6 cycles for the Flash (aT_F) and 1 cycle for SPM (aT_S). Additionally, we set the DMA transfer rate (D_{tr}) to 6.67 MB/s and the DMA arbitration time (D_{at}) to 2 cycles (STMicroelectronics 2022b). 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.

We evaluated benchmarks from `AutoBench`, `Networking`, and `TeleBench` benchmark suites offered by the `Embedded Microprocessor Benchmark Consortium (EEMBC)` (Poovey et al. 2009). `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. The evaluations presented in this section are performed using 20 benchmarks that are offered by `AutoBench`, `Networking`, and `TeleBench` benchmark suites. Table 3 provides some statistics about the

benchmarks from EEMBC. 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.

We used the following hyper-parameter values for FPA and SPEA algorithms. For FPA, as the probability of global pollination is always lower than local pollination in nature, the switch probability that switches update operators between the local and the global pollination is set to 0.8. Based on the parametric studies done by Yang et al. (2014) and Rodrigues et al. (2015), the scaling factor of 0.1 and $\gamma = 1.5$ for the standard gamma function works best in most cases. For SPEA, we use single-point crossover and multi-bit mutation strategies to update individuals

Table 3 Statistics for the benchmarks from EEMBC

Benchmarks	# Functions	# Loops	# Global Data Objects	Code Size	Data Size
Auto_a2time	4	1	9	2366	3056
Auto_aifftr	6	32	12	2748	29720
Auto_aifirf	3	13	24	1406	5436
Auto_aiifft	5	25	13	2136	29720
Auto_basefp	3	10	10	1180	18412
Auto_cacheb	19	2	31	2104	67260
Auto_canrdr	4	11	12	1500	13036
Auto_iirflt	3	13	32	2760	5340
Auto_pntrch	4	6	14	1150	6536
Auto_puwmod	3	1	8	1438	10732
Auto_rspeed	3	1	9	708	3056
Auto_tblook	4	8	18	1966	15192
Auto_ttsprk	8	5	60	3666	57764
Netw_ip_pktcheck	15	10	13	970	531461
Netw_ospfv2	8	9	6	720	51204
Tele_autocor	5	5	0	344	0
Tele_conven	2	7	0	312	0
Tele_fbital	2	6	0	350	0
Tele_fft	3	9	0	614	0
Tele_viterb	7	11	0	860	0

during recombination and mutation. We set the size of the set of the external population to 10, the recombination probability to 0.8, and the mutation probability to 0.2 (Zitzler 1999). For both algorithms, we set the population size for each generation to 10, the maximum number of generations to 50, and the number of initial populations $I = 5$ (cf. Algorithm (4), Line 3). Therefore, we perform a total of $(5 * 10 * 50) = 2500$ WCET and energy analysis each while solving a multi-objective optimization using either FPA and SPEA algorithm. As a multi-objective optimization is solved using both FPA and SPEA algorithm during these evaluations, a total of $(2 * 2 * 2500) = 10000$ evaluations are conducted for each benchmark. We need no extra parameter setting for ILP-based optimizations. The results obtained during these evaluations are valid for the algorithm parameters described above.

8.2 DMA call placement optimization (*DCPO*)

In this subsection, we compare the quality of FPA, SPEA, and ILP methods for the *DCPO* optimization problem. For this purpose, we run MO_{DMA} optimization to generate dynamically allocated individuals. The MO_{DMA} optimization is performed just to get the dynamically allocated individuals (x) that serve as input 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 DMA transfer in arbitration mode (cf. Eq. (16)) for each individual obtained using FPA, SPEA, and ILP methods, respectively.

Algorithm (4) describes the steps to solve the proposed MO_{DMA} problem. In this subsection, we use Algorithm (4) to generate a population consisting of individual solution vectors x, and for each x, Algorithm (3) is called to generate dynamically allocated code. To evaluate the quality of *DCPO* solutions, we solve the *DCPO* problem three times using FPA, SPEA, and ILP methods in this subsection, and collect $\mathcal{T}'_F$, $\mathcal{T}'_S$, and $\mathcal{T}'_I$ values. We select the worst DMA execution time value from $\mathcal{T}'_F$, $\mathcal{T}'_S$, and $\mathcal{T}'_I$, and that value will be eventually added to the WCET value of the solution vector x to advance the population generations in Algorithm (4). We insert DMA calls according to the worst *DCPO* solution, generate dynamically allocated code, and continue running Algorithms (3) and (4) to generate more individuals for evaluation.

For comparison and finding the best approach to solve *DCPO*, 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):

$$\mathrm{T}_\kappa = \left(1 - \frac{\mathcal{T}'_\kappa}{\sum_\kappa \mathcal{T}'_\kappa}\right) * 100 \tag{26}$$

where κ is the respective method and $\mathtt{T}_\kappa \in [0, 100]$. This is a standard way to normalize the value of a parameter (in our case $\mathcal{T}'_\kappa$) between the range [0, 100]. The $\mathtt{T}_\kappa$ metric quantifies if method $\kappa \in \{F, S, I\}$—FPA, SPEA, or ILP—is better relative to the other considered methods. The higher the value of $\mathtt{T}_\kappa$, the better the solution quality.

Let $\mathtt{T}_F$, $\mathtt{T}_S$, and $\mathtt{T}_I$ be 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 $\mathtt{T}_F$, $\mathtt{T}_S$, and $\mathtt{T}_I$ 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 $\mathtt{T}_F$, $\mathtt{T}_S$, and $\mathtt{T}_I$ metrics. The higher the median of the relative execution metric, the better the respective optimization method for *DCPO*.

Figure 8 shows the box plots for all benchmarks, where the relative execution metrics $\mathtt{T}_\kappa$ for FPA, SPEA, and ILP for all individuals from MO_{DMA} are plotted. The average $\mathtt{T}_\kappa$ value can skew due to outliers, so we use the median values. 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.

In Figure 8, 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 rest of the 7 benchmarks, the FPA- and SPEA-based approaches provided the same median value for the $\mathtt{T}_\kappa$ metric. Therefore, the SPEA algorithm performed slightly better than FPA during this evaluation run.

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 ($\mathcal{T}(x)$). 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 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.

8.3 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_{MEM}, SO_D, and SO_{DMA} optimizations (cf. Table 2), and compare all the optimization runs in terms of the quality of the obtained solutions. MO_{DMA}, MO_S, and MO_{MEM} simultaneously minimize WCET and

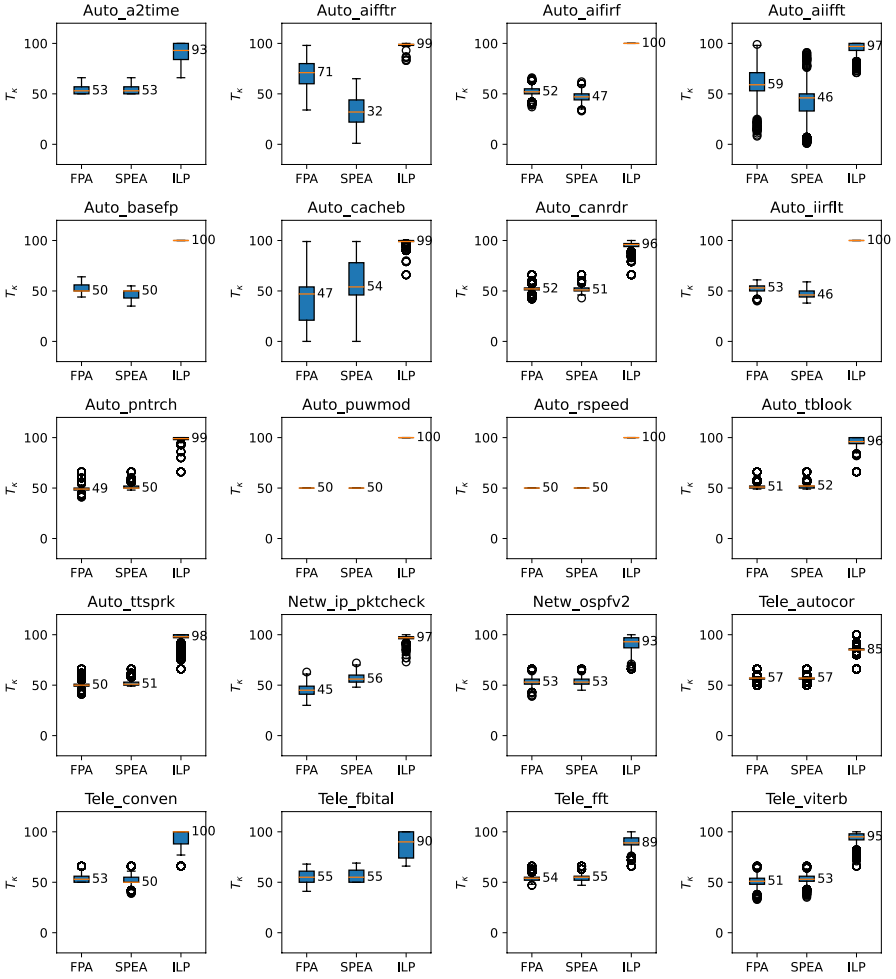


Fig. 8 Comparison between the relative execution metrics for *DCPO*–FPA, *DCPO*–SPEA, and *DCPO*–ILP

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. The *BaseLine* evaluation is performed by placing the complete program in the Flash memory, and we perform WCET and energy analysis to obtain the *BaseLine*'s respective objective values. Therefore, in the case of SO_D, SO_{DMA}, and *BaseLine*, we will always have one solution on their respective Pareto fronts.

As *DCPO*–FPA, on average, provided the worst solution quality in Sect. 8.2 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_{MEM} 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 *DCPO* method and MO_{DMA} will also provide better solutions.

8.3.1 Pareto front

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 approximated true Pareto front (PF_T) for each benchmark from the union of solutions obtained from all the optimization runs (Knowles et al. 2006). For example, let K and L be two Pareto fronts obtained from two optimization runs, then the final Pareto front PF_T, constructed from K and L, is the set of all non-dominated points from the union of the sets K and L, i.e.,

$$PF_T = \{a_i \mid \forall a_i \nexists b_j \in (K \cup L) \prec b_i\} \tag{27}$$

During this evaluation, PF_F is the final Pareto front obtained at the end of each optimization run, and 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_{MEM}–FPA, MO_{MEM}–SPEA, SO_D–ILP, SO_{DMA}–ILP, and the *Base-Line*, against which we evaluate the quality of solutions for each optimization.

Figure 9 shows the solutions found by MO_{DMA}, MO_S, and MO_{MEM} using FPA and SPEA, SO_D and SO_{DMA} using ILPs, and the *BaseLine*. For brevity's sake, we present the solutions for only four benchmarks[1]. In the figure, 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 `Auto_puwmod` benchmark, MO_{DMA}–FPA, MO_{DMA}–SPEA, MO_S–SPEA, and SO_{DMA} optimization runs contributed solutions to the final PF_T. MO_{DMA}–FPA, in this case, contributed the most 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 lie on

[1] The figures for all remaining benchmarks can be made available at the readers' request.

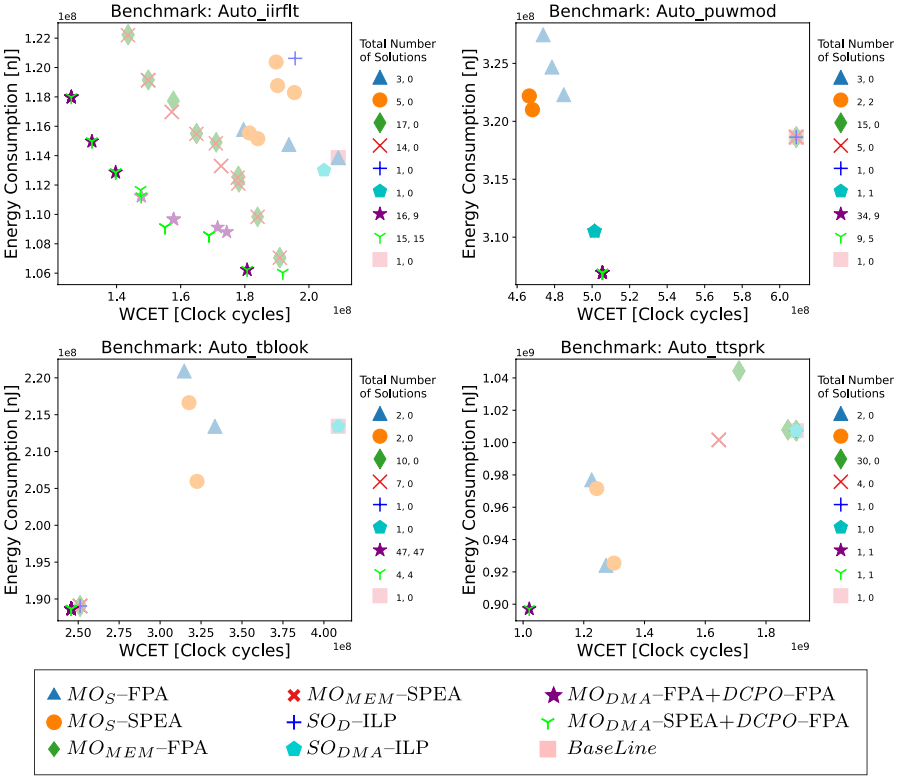


Fig. 9 Pareto front for `Auto_iirflt`, `Auto_puwmod`, `Auto_tblook`, and `Auto_ttsprk` benchmarks

the final PF_T. In the case of the `Auto_ttsprk` benchmark, both MO_{DMA}–FPA and MO_{DMA}–SPEA found one solution that lies on the final PF_T.

In some cases, it might happen that multiple solutions can be mapped to the same point in objective space. For example, for `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. For this benchmark, some data code objects did not impact its WCET and energy consumption. Therefore, the optimization produced a variety of optimal solutions with two points mapped 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 benchmarks. Table 4 provides the percentage of the number of solutions on PF_T on average out of the solution on PF_F for each optimization run. None of the solutions provided by MO_{MEM}, SO_D, and the *BaseLine* 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. For `Auto_rspeed`, the solution obtained by SO_{DMA} lied 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. Fig. 9). However, MO_{DMA} clearly provided most solutions on the final PF_T, and the FPA algorithm contributed slightly more solutions on PF_T than SPEA for MO_{DMA}.

In this paper, we do not address the process of selecting a solution from the Pareto front. After solving the multi-objective optimization problems, we might get many solutions on the final Pareto front, as illustrated in Fig. 9. The system designer is presented with the choice of multiple solutions after the metaheuristic algorithms have finished performing the optimization. Although this topic is beyond the scope of this paper, there are two potential approaches to dealing with the choice of a solution from the Pareto front:

1. We can add application-specific constraints to the optimization itself. It could further constrain the solutions 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 solutions that fulfill the provided constraint.
2. The system designer could use a so-called selector to select a solution from the pool of Pareto optimal solutions based on some runtime requirements.

Finally, we can conclude from the Table 4 and the plots of the four representative benchmarks in Fig. 9 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 compared to the *BaseLine*.
- MO_{MEM} is always clearly outperformed by MO_{DMA} due to the large overheads of CPU-based dynamic copying of code objects.
- Static allocation-based multi-objective optimization occasionally produces solutions of good quality on the Pareto front, but MO_{DMA}, in general, generates a much larger variation of final Pareto optimal solutions.

8.3.2 Quality metrics

To better analyze these trends, we consider four standard quality metrics–*Coverage* (*CV*), *Non-dominance Ratio* (*NR*), *Ratio of Non-dominated Solutions* (*RNS*), and *Hypervolume* (*HV*)—and compare all the optimizations using them.

Definition 7 (*Coverage*) *Coverage* (*CV*) describes the total number of dominated points in a solution set PF_F (Zitzler 1999).

Table 4 Comparison between the average number of solutions in percentage 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_{MEM}		SO_D	SO_{DMA}	MO_{DMA}		*BaseLine*
	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
Average # Solutions on PF_o (%)	9.09%	47.27%	0%	0%	0%	10.56%	73.8%	71.59%	0%

$$CV = 1 - \frac{|\{a \in PF_F : \exists b \in PF_T, a \preceq b\}|}{|PF_F|} \in [0, 1] \tag{28}$$

If all the solutions on PF_T dominate PF_F, then $CV = 1$, and if all solutions on PF_F dominate or are equal to the ones on PF_T, then $CV = 0$. The lower the value of *CV*, the better.

Definition 8 (*Non-Dominance Ratio*) *Non-Dominance Ratio* (*NR*) is the ratio of non-dominated solutions contributed by PF_F to the total non-dominated solutions in PF_T (Goh and Tan 2008). The higher the value of *NR*, the better.

$$NR = \frac{|PF_T \cap PF_F|}{|PF_T|} \in [0, 1] \tag{29}$$

Definition 9 (*Ratio of Non-dominated Solutions*) *Ratio of Non-dominated Solutions*(*RNS*) is the ratio of the number of non-dominated solutions on PF_F to the total number of solutions on PF_F itself (Tan et al. 2002). The higher the value of *RNS*, the better.

$$RNS = \frac{|\{a \in PF_F : a \in PF_T\}|}{|PF_F|} \in [0, 1] \tag{30}$$

Definition 10 (*Hypervolume*) *Hypervolume* (*HV*) measures the volume of the objective space dominated by the non-dominated solutions with respect to an appropriately chosen reference point (**R**), i.e.,

$$HV = \lambda(\mathbb{D}(PF_F)) \tag{31}$$

where $\lambda(.)$ denotes the Lebesgue 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 **R** (Zitzler 1999; Kuhn et al. 2016).

For a minimization problem, **R** has objective coordinates greater than all the solutions in the Pareto front. Moreover, in our case, the objective space is 2-dimensional ($q = 2$), and *HV* calculates the area of the region bounded by the reference point and the Pareto front. The higher the area enclosed by the Pareto front, the better the *HV* metric.

Table 5 summarizes the total number of benchmarks with best or nondominated *CV*, *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 metrics, 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. Fig. 9). In these cases, we cannot conclude the best approach as multiple approaches provided solutions on the final PF_T. For the `Auto_basefp` benchmark, MO_{DMA}–FPA found 4 solutions on its PF_F, and all 4 solutions were the only solutions on the final PF_T. Therefore, in this case, we can say that MO_{DMA}–FPA performed best compared to other approaches.

Table 5 shows that, on average, MO_{DMA} provided better quality solutions compared to previous approaches. MO_{MEM}, SO_D, and the *BaseLine* provided no solutions on the final PF_T for any benchmark. SO_{DMA} provided a nondominated solution in terms of *CV* and *RNS* for two benchmarks and for one benchmark in terms of *HV*. For some benchmarks, MO_S provided best or nondominated solutions on PF_T.

For the `Auto_aiifft` benchmark, MO_S–SPEA provided all the solutions on the final PF_T. For `Tele_autocor` and `Tele_conven`, 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 `Netw_ip_pktcheck`, MO_S–FPA and MO_S–FPA provided nondominated solutions, with most solutions provided by MO_S–SPEA.

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 metrics than MO_S–FPA.

These benchmarks had functions and loops bigger than *ISPM* size, making basic block-level MO_S more effective for these benchmarks. Therefore, a system designer can choose to use MO_{DMA} if the following condition is satisfied:

$$\sum_{p=1}^{P} \sum_{c=1}^{C_p} S_{\mathcal{C}_{(p,c)}} \geq S_{ISPM} \quad \forall S_{\mathcal{C}_{(p,c)}} \leq S_{ISPM} \tag{32}$$

where $S_{\mathcal{C}_{(p,c)}}$ is the size of the code object $\mathcal{C}_{(p,c)}$, and S_{ISPM} is the size of the *ISPM*. Equation (32) implies that the sum of the sizes of the code objects smaller than *ISPM* size must be greater or equal to *ISPM* size. If this condition is not satisfied, MO_S might provide better-quality solutions.

Moreover, SO_{DMA} performs function-level dynamic allocation and uses the DMA controller in burst mode to perform DMA transfers. As most benchmarks in `EEMBC`

contain some large functions that were not fully able to fit in the SPM, SO_{DMA} ended up placing small functions within the SPM that did not contribute too much to the overall WCET reduction of the program. 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}.

For 2 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 *CV* and *RNS*. However, MO_{DMA}–FPA outperformed the other two approaches in terms of *NR* metric by providing the highest number of solutions to the final PF_T. In terms of *HV*, MO_{DMA}–FPA and MO_{DMA}–SPEA provided nondominated solutions on PF_T for these two benchmarks.

For `Auto_rspeed`, MO_{DMA}–FPA, MO_{DMA}–SPEA, SO_{DMA}, and MO_S–SPEA provided nondominated solutions in terms of *CV* and *RNS*. MO_{DMA}–FPA, MO_{DMA} –SPEA, and SO_{DMA} provided nondominated solutions in terms of *HV*, and only MO_{DMA}–FPA provided the most number of solutions with best *NR* value. For `Netw_ospfv2`, both MO_{DMA}–FPA and MO_{DMA}–SPEA provided one solution each, and MO_S–SPEA provided two solutions on the final PF_T.

For the rest of the benchmarks, MO_{DMA} clearly outperformed all other approaches. For 10 and 9 benchmarks, MO_{DMA}–FPA and MO_{DMA}–SPEA have nondominated performance in terms of *CV* and *RNS*, with 3 and 4 benchmarks having nondominated *NR* metrics, respectively. Moreover, for 10 benchmarks, MO_{DMA}–FPA and MO_{DMA}–SPEA have nondominated *HV* metric. For 3 benchmarks, MO_{DMA}–SPEA provided the best solutions in terms of *HV* metric. Finally, for 2 benchmarks, MO_{DMA}–FPA and MO_{DMA}–SPEA each provided the best solutions in terms of *CV* and *RNS*, with 6 and 4 benchmarks having the best *NR* metrics, respectively. Overall, MO_{DMA}–FPA provided more solutions on the final PF_T than MO_{DMA}–SPEA. When compared to the rest of the approaches, our proposed MO_{DMA} approach is clearly better when Eq. (32) is satisfied.

8.3.3 Compilation times

MO_{DMA} is a time-consuming optimization as it needs us to perform WCET and energy analyses, solve address assignment problems, and perform *DCPO* multiple times. 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 extremely fast than MO_{DMA} at the determent of SO_D and SO_{DMA}'s solution quality.

SO_{DMA} and SO_D , on average, took 8.02 min and 26.08 min to finish the optimization runs, respectively. MO_S–FPA and MO_S–SPEA, on average, took 52.68 h and 45.48 h to finish the optimization runs, respectively. MO_{MEM} –FPA and MO_{MEM}–SPEA took 48.89 h and 44.07 h, on average, to finish the optimization runs, respectively. MO_{DMA}–FPA and MO_{DMA}–SPEA took 69.72 h and 69.11 h, on average, to finish the optimization runs, respectively.

In Sect. 8.1, we calculated the maximum number of WCET and energy evaluations to complete the multi-objective optimization using metaheuristic

Table 5 Total number of benchmarks with best (or nondominated) quality metrics for each optimization approach for 60% relative SPM size and MO_{DMA}+*DCPO*–FPA configuration

		MO_S		MO_{MEM}		SO_D	SO_{DMA}	MO_{DMA}		*BaseLine*
		FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
Total number of Benchmarks with best (or nondominated) quality metrics:	*CV*	0 (2)	3 (7)	0 (0)	0 (0)	0 (0)	0 (2)	2 (10)	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 (0)	0 (2)	2 (10)	2 (9)	0 (0)
	HV	1 (0)	6 (0)	0 (0)	0 (0)	0 (0)	0 (1)	0 (10)	3 (10)	0 (0)

algorithms. Therefore, as MO_{DMA}, MO_{MEM}, and MO_S optimization runs require a maximum of 10000 evaluations for each benchmark, they are more time-consuming than the single-objective optimizations during these evaluations. Moreover, MO_{DMA} needs to perform not only WCET and energy analyses but also solve the address assignment problems and perform *DCPO* multiple times. 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_{MEM} and MO_S, respectively.

The multi-objective optimization MO_{DMA} proposed in this paper is time-consuming. The requirement of multiple objective analyses acts as a compilation time bottleneck. For real-world applications, MO_{DMA} can become more time-consuming based on the time required for the respective analyses. Therefore, in the future, it is necessary to focus on developing approaches that can quickly approximate WCET and energy consumption values for dynamically allocated code. We can replace the costly WCET and energy analyses with such approximation models to mitigate the compilation time bottleneck.

8.3.4 Objective comparisons

We compared MO_{DMA} with MO_S, MO_{MEM}, SO_D, SO_{DMA}, and the *BaseLine* evaluations to determine the average reduction in objective values achieved by MO_{DMA}. Table 6 presents the average reductions in WCET and energy consumption obtained by MO_{DMA}+*DCPO*–FPA, in comparison to the other approaches.

As noted in previous subsections, MO_S performed better than MO_{DMA} for some benchmarks, which is reflected in the average reduction values as well. While MO_{DMA} showed a lesser overall reduction in WCET and energy consumption compared to MO_S, it still performed better on average than MO_S across all benchmarks. In comparison to the other approaches, MO_{DMA} demonstrated a clear advantage. Additionally, when compared to the *BaseLine*, MO_{DMA} achieved an average improvement of 39.85% in WCET and 10.62% in energy consumption.

8.4 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 we solve the MO_{DMA} problem using *DCPO*–ILP instead of *DCPO*–FPA. We compare the solutions obtained from MO_{DMA}+*DCPO*–ILP optimization run with solutions obtained from all other approaches. Figure 10 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. Comparing Figs. 10 and 9 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

Table 6 Average reductions in WCET and energy consumption (in %) obtained using MO_{DMA}+*DCPO*–FPA over MO_S, MO_{MEM}, SO_D, SO_{DMA}, and the *BaseLine*

Objectives	MO_S	MO_{MEM}	SO_D	SO_{DMA}	*BaseLine*
WCET	11.61%	20.34%	24.54%	30.07%	39.85%
Energy consumption	3.43%	7.79%	10.58%	10.14%	10.62%

`Auto_ttsprk`, MO_{DMA}–FPA+*DCPO*–ILP was the only optimization that found the single solution on the final Pareto front PF_T.

Due to the random nature of the metaheuristic algorithms, it is impossible to predict the nature and the shape of the final Pareto front. It is evident from Figs. 10 and 9 that, for the same benchmarks, we end up with different-looking Pareto fronts. However, the trend that MO_{DMA} performed better than other approaches for most benchmarks remained the same. Furthermore, due to *DCPO*–ILP instead of *DCPO*–FPA, we were able to find slightly better solutions using MO_{DMA} in this subsection compared to Sect. 8.3.

Table 7 provides the total number of benchmarks with best or nondominated Coverage (*CV*), Non-dominance Ratio (*NR*), Ratio of Non-dominated Solutions (*RNS*), and Hypervolume (*HV*) metrics for each approach, and it also provides a comparison between the average number of solutions obtained by each approach on the final PF_T from respective PF_F. Comparing Tables 7 and 5, we can see that there is variation in the number of best and nondominated quality metrics. Overall, MO_{DMA}+*DCPO*–ILP provided slightly better solutions in this optimization run compared to the previous MO_{DMA}+*DCPO*–FPA run. Moreover, MO_{DMA}+*DCPO*–ILP has more solutions on the final PF_T than in Sect. 8.3.

MO_{DMA} performed better than other approaches with the best or nondominated quality metrics for most benchmarks. MO_{DMA}–SPEA performed best for 3 benchmarks in terms of *CV* and *RNS* metrics, and MO_{DMA}–FPA performed best for 8 benchmarks in terms of *NR* quality metric. MO_S–SPEA provided 6 benchmarks with better *HV* metric. However, overall MO_{DMA} had better or nondominated *HV* values for more benchmarks. Both, MO_{DMA}–FPA and MO_{DMA}–SPEA performed equally well in terms of *HV* metric. Overall, MO_{DMA}–SPEA performed better in terms of *CV* 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 final PF_T and MO_{DMA}–FPA had most (86.75%) solutions on the final PF_T.

We compared the performance of our proposed MO_{DMA}+*DCPO*–ILP approach with MO_S, MO_{MEM}, SO_D, SO_{DMA}, and the *BaseLine* evaluation in terms of objective values. Table 8 presents the average reductions in WCET and energy consumption achieved by MO_{DMA}+*DCPO*–ILP in comparison to the other approaches. By examining Tables 6 and 8, we can clearly observe similar trends, with MO_{DMA} showing greater reduction in both WCET and energy consumption compared to the other approaches.

We also compared the compilation time required by these approaches, and similar to Sect. 8.3, in this subsection, SO_D and SO_{DMA} provided poor solution quality

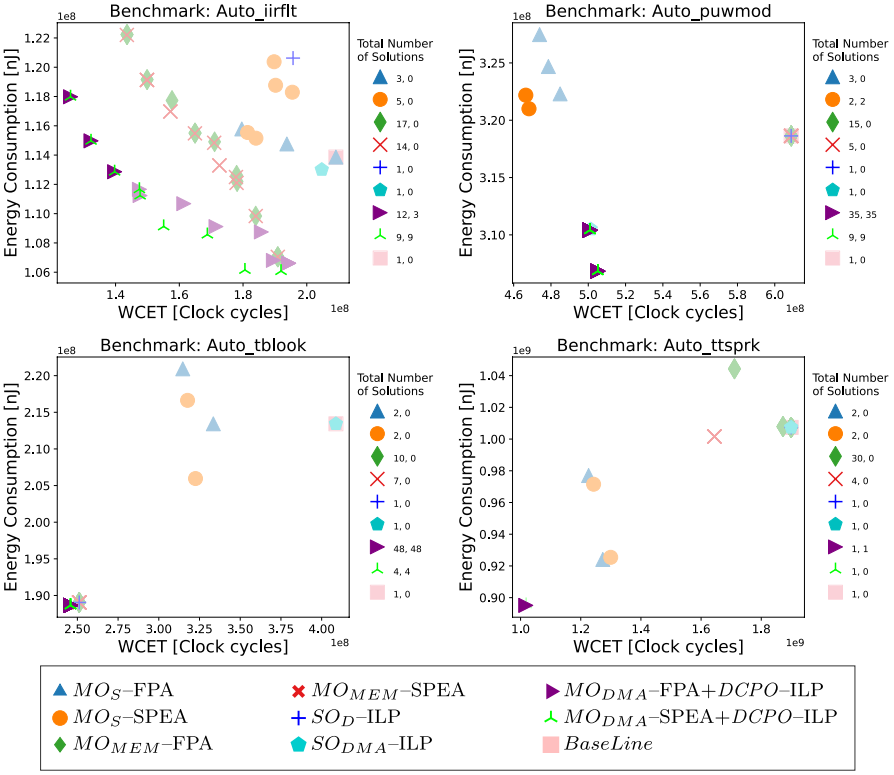


Fig. 10 Pareto front for `Auto_iirflt`, `Auto_puwmod`, `Auto_tblook`, and `Auto_ttsprk` benchmarks

but were extremely fast than MO_{DMA}. Moreover, MO_{DMA}+$DCPO$–ILP, on average, needed 27.57% more compilation time than MO_S. However, MO_{DMA}+$DCPO$–ILP was 4.17% faster than MO_{MEM}, i.e., MO_{DMA}+$DCPO$–ILP, on average, required 18.74% less compilation time than MO_{DMA}+$DCPO$–FPA. Solving the $DCPO$ problem contributes to the total compilation time for MO_{DMA}. However, as MO_{DMA}+$DCPO$–ILP was faster than MO_{MEM}, we can deduce in this case that the time taken to solve the $DCPO$–ILP problem is not the major contributing factor for MO_{DMA}+$DCPO$–ILP's compilation time. Generally, the compilation time required by MO_S, MO_{MEM}, 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 conclude trends for compilation times for these multi-objective optimizations.

From the evaluations conducted in Sects. 8.2, 8.3, and 8.4, 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.

Table 7 Total number of benchmarks with best (or nondominated) quality metrics and comparison between the average number of solutions in percentage on the final PF_T from respective PF_F for each optimization approach for 60% relative SPM size and MO_{DMA}+*DCPO*–ILP configuration

		MO_S		MO_{MEM}		SO_D	SO_{DMA}	MO_{DMA}		*BaseLine*
		FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
Total number of Benchmarks with best (or nondominated) quality metrics:	*CV*	0 (2)	3 (6)	0 (0)	0 (0)	0 (0)	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)	3 (6)	0 (0)	0 (0)	0 (0)	0 (1)	2 (10)	3 (10)	0 (0)
	HV	1 (0)	6 (0)	0 (0)	0 (0)	0 (0)	0 (1)	2 (9)	2 (9)	0 (0)
Average # Solutions on PF_o (%)		9.09%	47.27%	0%	0%	0%	5.26%	86.75%	78.57%	0%

Table 8 Average reductions in WCET and energy consumption (in %) obtained using MO_{DMA}+*DCPO*–ILP over MO_S, MO_{MEM}, SO_D, SO_{DMA}, and the *BaseLine*

Objectives	MO_S	MO_{MEM}	SO_D	SO_{DMA}	*BaseLine*
WCET	10.81%	19.17%	24.83%	30.32%	39.57%
Energy consumption	3.48%	7.64%	10.51%	10.59%	11.01%

- Due to the random nature of the metaheuristic algorithms, it is difficult to consistently get the same solutions on the final PF_T. However, MO_{DMA} with both worst-performing *DCPO*–FPA and best-performing *DCPO*–ILP provides better solutions compared to all other approaches, except when Eq. (32) is not satisfied, MO_S can provide better solutions.
- MO_{DMA} provided a significant reduction in WCET and energy consumption, especially compared to the *BaseLine* evaluation.
- MO_{DMA} outperformed all previous dynamic allocation-based optimization approaches.
- Using DMA instead of CPU to perform dynamic allocation during runtime clearly reduces the WCET and energy consumption of the program under consideration.

8.5 Optimizations comparison with 20%, 40%, and 80% relative SPM size

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 MO_{DMA} optimization in conjunction with *DCPO*–FPA.

Table 9 provides the total number of benchmarks with best or nondominated Coverage (*CV*), Non-dominance Ratio (*NR*), Ratio of Non-dominated Solutions (*RNS*), and Hypervolume (*HV*) metrics for each approach for 20%, 40%, and 80% relative SPM sizes. Similar to Table 5, in Table 9, the numbers shown outside the bracket represent the total number of benchmarks with best quality metrics, and the numbers inside the bracket represent the number of benchmarks with nondominated Pareto optimal solutions compared to other approaches.

MO_S–SPEA provided best solutions in terms of *NR* for a total of 3, 2, and 4 benchmarks in 20%, 40%, and 80% relative SPM size cases, respectively, and two benchmarks with best solutions in terms of *CV* and *RNS* in all SPM size cases. MO_S–FPA performed best for only one benchmark in terms of *CV* and *RNS* in 80% relative SPM size case. MO_S–SPEA and MO_S–FPA provided best solution in terms of *HV* metric for 3 and 4 benchmarks in 80% relative SPM size case. For the multi-objective static allocation approach, MO_S–SPEA performed better than MO_S–FPA in terms of all quality metrics for all SPM size configurations.

MO_{MEM}–SPEA was able to provide nondominated solutions for only one benchmark in terms of *CV* and *RNS* in 40% SPM size case, and for the rest of the cases, MO_{MEM}–SPEA contributed no solutions to the final Pareto front. SO_{DMA} provided the best solution in terms of all quality metrics only for one benchmark in the 20%

Table 9 Total number of benchmarks with best (or nondominated) quality metrics for each optimization approach for 20%, 40%, and 80% relative SPM size configuration

Relative	Quality	MO_S		MO_{MEM}		SO_D	SO_{DMA}	MO_{DMA}		*BaseLine*
SPM size (%)	metrics	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
20%	*CV*	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)	1 (6)	9 (6)	0 (0)
40%	*CV*	0 (1)	2 (2)	0 (0)	0 (1)	0 (0)	0 (0)	5 (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)	0 (0)	0 (0)	5 (7)	4 (7)	0 (0)
	HV	0 (0)	0 (1)	0 (0)	1 (0)	0 (0)	0 (0)	4 (9)	5 (9)	0 (0)
80%	*CV*	1 (4)	2 (2)	0 (0)	0 (0)	0 (0)	0 (1)	3 (9)	2 (9)	0 (0)
	NR	0 (3)	4 (2)	0 (0)	0 (0)	0 (0)	0 (0)	9 (3)	2 (2)	0 (0)
	RNS	1 (4)	2 (2)	0 (0)	0 (0)	0 (0)	0 (1)	3 (9)	2 (9)	0 (0)
	HV	3 (0)	4 (0)	0 (0)	0 (0)	0 (0)	1 (0)	2 (8)	2 (8)	0 (0)

SPM size case. Moreover, SO_{DMA} provided a nondominated solution for one benchmark in terms of *CV* and *RNS*, and one best solution in terms of *HV* metric in the 80% relative SPM size case. MO_{MEM}–FPA and SO_D did not provide any solutions on the final Pareto front for any benchmark in an SPM size configuration.

From the overall table, we can clearly see that MO_{DMA} provided better quality solutions for most benchmarks on average compared to all other approaches. For 20% relative SPM size, MO_{DMA}–SPEA provided better solutions in terms of *CV*, *RNS*, and *HV*, and MO_{DMA}–FPA was better in terms of *NR*, i.e., MO_{DMA}–FPA produced the most 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 more for MO_{DMA}–SPEA.

The behavior of MO_{DMA} for 40% and 80% relative SPM sizes is similar to the 60% SPM size case described in Sect. 8.3. On average, MO_{DMA} provided better quality solutions compared to previous approaches for most benchmarks. MO_{DMA}–FPA performed better when compared to MO_{DMA}–SPEA in terms of *CV*, *NR*, and *RNS* metrics. However, both MO_{DMA}–FPA and MO_{DMA}–SPEA performed almost equally well in terms of *HV* metric.

Table 10 provides the percentage of the number of solutions on the final PF_T, on average, out of the solutions on 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 most number of solutions on the final Pareto front. For 20% SPM size, MO_{DMA}–SPEA produced the most solutions on the final Pareto front, and for the rest of SPM configurations, MO_{DMA}–FPA had the most number of solutions on the final Pareto front.

In the case of MO_S, we can see that the percentage of solutions contributed on 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. MO_{MEM} (20% and 40% SPM size) and SO_{DMA} (20% and 80% SPM size) contributed some solutions on the final Pareto front. Finally, SO_D and the *BaseLine* contributed no solutions in any configuration.

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—as evident from Table 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 only 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 Eq. (32), 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_{MEM}, SO_D, and SO_{DMA}.

Table 11 presents the average percentage reduction in WCET and energy consumption achieved using MO_{DMA} compared to MO_S, MO_{MEM}, SO_D, SO_{DMA}, and the *BaseLine* evaluation. The reduction in WCET and energy consumption depends on the number of code 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 MO_{DMA}. But, in general, if we assume that the bigger the size of SPM, the more code 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 code objects that cannot be dynamically allocated either due to their size or liveness conflicts, in that case, the reduction in WCET and energy objectives will be comparatively less. In either scenario, we can conclusively say that MO_{DMA} outperforms MO_{MEM} for any SPM size configuration.

However, in the case of MO_S, the larger the size of SPM, the smaller the reduction in WCET and energy consumption. As MO_S can place more basic blocks due to increased SPM size, this behavior is expected. On the contrary, the larger the size of the SPM, the more reduction in WCET and energy consumption is achieved by MO_{DMA} compared to the *BaseLine* evaluation.

8.6 Preliminary tests on hardware

During these evaluations, we also considered performing some tests on actual hardware. As it would be extremely difficult to replicate the worst-case scenario on actual hardware, we will get final solutions with an execution time that is always less than WCET. Consequently, we cannot exactly match the reduction in the analyzed WCET and the reduction in the measured execution time. However, we performed preliminary tests to execute the compiled code on actual hardware. For the STM32F0-Discovery (2012) board, we were not able to use the UART (Universal Asynchronous Receiver Transmitter) over the USB port. Therefore, we used the STM32F091 Nucleo-64 (2020) board to perform these preliminary tests.

Table 10 Comparison between the average number of solutions in percentage 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_{MEM}		SO_D	SO_{DMA}	MO_{DMA}		*BaseLine*
SPM size (%)	FPA	SPEA	FPA	SPEA	ILP	ILP	FPA	SPEA	
20%	1.59%	6.58%	0.46%	1.03%	0%	5%	63.57%	70.33%	0%
40%	2.74%	11.86%	0%	1.19%	0%	0%	75.75%	68.18%	0%
80%	28.95%	32.43%	0%	0%	0%	5.26%	72.76%	66.23%	0%

Table 11 Average reduction in WCET and energy consumption (in %) obtained using MO_{DMA} over MO_S, MO_{MEM}, SO_D, SO_{DMA}, and the *BaseLine* for 20%, 40%, and 80% relative SPM size configuration

Relative SPM Size(%)	Objectives	MO_S	MO_{MEM}	SO_D	SO_{DMA}	*BaseLine*
20%	WCET	21.69%	15.92%	26.8%	27.45%	32.23%
	Energy consumption	4%	4.43%	11.5%	10.2%	8.65%
40%	WCET	20.07%	18.39%	25.28%	31.78%	38.97%
	Energy consumption	5.72%	7.49%	10.7%	10.17%	10.75%
80%	WCET	6.72%	21.52%	28.13%	32.6%	43.61%
	Energy consumption	2.92%	6.74%	11.85%	10.54%	12.91%

For this preliminary setup, we used the WCC compiler to generate code and STM32CubeIDE (2025) to flash and monitor the generated code on the Nucleo board. We used a built-in timer (TIM2) to measure the execution time of the code, UART2 to display the execution times via PuTTY, and DMA Channel 1 for memory-to-memory transfer.

With this preliminary setup, we were able to statically allocate code to the SPM during startup and measure the execution times for statically allocated code. Using this setup, we encountered hard fault interrupts for purely dynamically allocated code. However, we are able to execute a “pseudo" dynamically allocated code with this hardware setup, i.e., for example, let us consider a code object $\mathcal{C}_{(1,1)}$ that we want to dynamically allocate to SPM during runtime. In the pseudo dynamically allocated code, we have two copies of the code object $\mathcal{C}_{(1,1)}$: one is in the Flash at address $\alpha_{(1,1,1)}$, and another is statically allocated to SPM at address $\delta_{(1,1,1)}$ during startup. During runtime, we dynamically copy code object $\mathcal{C}_{(1,1)}$ from $\alpha_{(1,1,1)}$ to $\delta_{(1,1,1)}$ using the DMA controller, i.e., the copy of $\mathcal{C}_{(1,1)}$ is overwritten in the SPM, and the code object $\mathcal{C}_{(1,1)}$ is executed from the SPM.

In the case of purely dynamically allocated code, the DMA controller can dynamically allocate the code object $\mathcal{C}_{(1,1)}$ from Flash to SPM. We can observe the dynamically copied code in SPM using the STM32CubeIDE interface. However, when the execution reaches $\mathcal{C}_{(1,1)}$ in SPM, we encounter a hard fault interrupt. This preliminary hardware setup either cannot realize that the SPM is dynamically populated during runtime, or the STM32CubeIDE is unable to recognize the debug information for the dynamically allocated code in SPM in its debug mode. Therefore, this

preliminary setup is not an appropriate setup to execute and measure the dynamically allocated code at the hardware level.

It is important to note that all the evaluations presented in previous subsections are performed using the compiler-level setup discussed in Sect. 8.1. The STM32CubeIDE issue is specific to the hardware setup discussed in this subsection and does not stem from any concepts or optimizations proposed in this paper. Consequently, the evaluations and comparisons presented in all the previous subsections of this paper are still correct and valid.

This paper primarily focuses on compiler-level mathematical modeling and optimization; therefore, performing hardware verifications is not the main emphasis. For brevity's sake, expanding the hardware setup for DMA-aware dynamically allocated code can be a future topic of exploration. In the future, we plan to develop a comprehensive setup that enables the measurement of dynamically allocated code at the hardware level. Furthermore, this hardware setup can also be expanded to perform DMA controller-related energy measurements in the future.

9 Conclusions

In this paper, we integrated the DMA model and the DMA analyzer within a compiler, and we proposed a DMA Call Placement Optimization (*DCPO*) that optimizes the total execution time required for DMA transfer through 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 single-objective *DCPO* problem.

We also proposed a DMA-aware dynamic SPM allocation-based multi-objective optimization (MO_{DMA}) that dynamically allocates code objects from the Flash memory to SPM using DMA 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 to compare our proposed approach with previously proposed techniques. We evaluated the performance of the proposed MO_{DMA} optimization by comparing the quality of its solutions with those obtained using MO_S, MO_{MEM}, SO_D, and SO_{DMA}. The single-objective optimizations (SO_D and SO_{DMA}) and the *Base-Line* failed to compete with MO_{DMA}, and due to large WCET and energy overheads required for CPU-based dynamic allocation, MO_{MEM} was outperformed by MO_{DMA}. Due to the difference between the granularity of the basic block- and code object-level allocation, static allocation-based multi-objective optimization 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 memory size to which 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. Our proposed approach outperformed MO_{MEM}, SO_D, SO_{DMA}, and the *BaseLine* 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 with an increase in SPM size. However, on average, our proposed approach MO_{DMA} performed better than other approaches for all SPM size configurations.

Dynamic allocation techniques require jump correction and the addition of extra code to facilitate dynamic allocation, leading to an increase in the code size. For example, during evaluations for MO_{DMA}+*DCPO*–FPA in Sect. 8.3, we observed, on average, a 47% increase in code size compared to the original benchmark size. Consequently, we aim to explore methods for integrating code compression into multi-objective dynamic allocation techniques in our future research, with a particular focus on considering code size as an important objective.

The multi-objective optimizations in this paper generate a large number of Pareto optimal solutions with visible trade-offs between multiple objectives. A possible future work could involve the design of a decision-maker based on specified criteria that help the system designer weed out, sort, or choose necessary solutions during runtime. Additionally, this paper does not address multi-task or multi-core systems. Future work can expand the proposed MO_{DMA} to handle multi-task and multi-core systems and consider the schedulability of such systems in the context of dynamic memory allocation. However, while executing dynamically allocated code from SPM, if the task gets preempted, it might present many challenges and many avenues for future work.

Unlike the execution time costs, the *DCPO* problem proposed in this paper does not address the energy consumption costs due to arbitration and CPU stalls during DMA transfers. Future work can focus on expanding the *DCPO* problem using an extended energy model that considers energy consumption due to a DMA controller. Moreover, we could expand the *DCPO* problem to perform a multi-objective optimization that aims to simultaneously minimize both the execution time and the energy consumption due to the placement of DMA transfer calls.

Furthermore, solving the MO_{DMA} problem can be time-consuming due to the need for multiple WCET and energy analyses, solving address assignment problems, and performing *DCPO*. In the future, we plan to investigate strategies to approximate objectives, reducing the number of evaluations required to bring MO_{DMA} to fruition and consequently decreasing the required compilation time.

Acknowledgements This work is part of a project that received funding from NXP Semiconductors.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability The evaluations results used to generate the figures and tables can be made available at the readers' request.

Declarations

financial interest or non-financial interest All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

References

AbsInt Angewandte Informatik, GmbH (2024) aiT worst-case execution time analyzers

Appel AW (2004) Modern compiler implementation in C. Cambridge University Press, Cambridge

Balasundaram A, Chenniappan V (2015) Optimal code layout for reducing energy consumption in embedded systems. In: 2015 international conference on soft-computing and networks security (ICSNS), IEEE, pp 1–5

Dasygenis M, Brockmeyer E, Durinck B et al (2006) A combined dma and application-specific prefetching approach for tackling the memory latency bottleneck. IEEE Trans Very Large Scale Integr (VLSI) Syst 14(3):279–291

Deverge JF, Puaut I (2007) WCET-directed dynamic scratchpad memory allocation of data. In: Euromicro conference on real-time systems (ECRTS), pp 179–190

Ehrgott M (2005) Multicriteria optimization, vol 491. Springer science & business media

Emmerich MT, Deutz AH (2018) A tutorial on multiobjective optimization: fundamentals and evolutionary methods. Natural Comput 17:585–609

Falk H, Kleinsorge JC (2009) Optimal static wcet-aware scratchpad allocation of program code. In: Proceedings of the 46th annual design automation conference, pp 732–737

Falk H, Lokuciejewski P (2010) A compiler framework for the reduction of worst-case execution times. Real-Time Syst 46(2):251–298

Francesco P, Marchal P, Atienza D et al (2004) An integrated hardware/software approach for run-time scratchpad management. In: Design automation conference (DAC), pp 238–243

Goh CK, Tan KC (2008) A competitive-cooperative coevolutionary paradigm for dynamic multiobjective optimization. IEEE Trans Evolut Comput 13(1):103–127

Gurobi Optimization Inc (2024) Gurobi optimizer. https://www.gurobi.com

Ishitobi Y, Ishihara T, Yasuura H (2007) Code placement for reducing the energy consumption of embedded processors with scratchpad and cache memories. In: Workshop on embedded systems for real-time multimedia, IEEE, pp 13–18

Jadhav S, Falk H (2019) Multi-objective optimization for the compiler of real-time systems based on flower pollination algorithm. In: Software and compilers for embedded systems (SCOPES). ACM, pp 45–48

Jadhav S, Falk H (2022) Approximating WCET and energy consumption for fast multi-objective memory allocation. In: Real-time networks and systems (RTNS), pp 162–172

Jadhav S, Falk H (2023) Towards multi-objective dynamic SPM allocation. In: Workshop on worst-case execution time analysis (WCET), DROPS

Kandemir M, Ramanujam J, Irwin MJ et al (2001) Dynamic management of scratch-pad memory space. In: Design automation conference (DAC), pp 690–695

Kim Y, Broman D, Shrivastava A (2017) WCET-aware function-level dynamic code management on scratchpad memory. ACM Trans Embed Comput Syst (TECS) 16(4):1–26

Knowles JD, Thiele L, Zitzler E (2006) A tutorial on the performance assessment of stochastic multiobjective optimizers. Tik report 214

Kuhn T, Fonseca CM, Paquete L et al (2016) Hypervolume subset selection in two dimensions: formulations and algorithms. Evolut Comput 24(3):411–425

Lokuciejewski P, Plazar S, Falk H et al (2011) Approximating pareto optimal compiler optimization sequences–a trade-off between WCET, ACET and code size. Softw: Pract Exp 41(21):1437–1458

Mancuso R, Dudko R, Caccamo M (2014) 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, IEEE, pp 1–10

Muchnick SS (1998) Advanced compiler design and implementation, 1st edn. Morgan Kaufmann Publishers Inc

Muts K, Falk H (2020) Multi-criteria function inlining for hard real-time systems. In: Real-time networks and systems (RTNS), pp 56–66

Nikov K, Georgiou K, Chamski Z et al (2022) 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). IEEE, pp 1–4

Oehlert D, Luppold A, Falk H (2016) Practical challenges of ILP-based SPM allocation optimizations. In: International workshop on software and compilers for embedded systems, ACM, pp 86–89

Pellizzoni R, Betti E, Bak S et al (2011) A predictable execution model for cots-based embedded systems. In: 2011 17th IEEE real-time and embedded technology and applications symposium, IEEE, pp 269–279

Poovey JA, Conte TM, Levy M et al (2009) A benchmark characterization of the EEMBC benchmark suite. IEEE Micro 29(5):18–29

Qiu M, Chen Z, Ming Z et al (2017) Energy-aware data allocation with hybrid memory for mobile cloud systems. IEEE Syst J 11(2)

Rodrigues D, Yang XS, De Souza AN et al (2015) Binary flower pollination algorithm and its application to feature selection. In: Recent advances in swarm intelligence and evolutionary computation. Springer, pp 85–100

Soliman MR, Pellizzoni R (2017) WCET-driven dynamic data scratchpad management with compiler-directed prefetching. In: Euromicro conference on real-time systems (ECRTS)

STM32CubeIDE (2025) UM2609 STM32CubeIDE User Guide. https://www.st.com/resource/en/user_manual/dm00629856.pdf

STM32F0-Discovery (2012) UM1525 STM32F0DISCOVERY user manual. https://www.st.com/resource/en/user_manual/um1525-stm32f0discovery-discovery-kit-for-stm32-f0-microcontrollers-stmicroelectronics.pdf

STM32F091 Nucleo-64 (2020) UM1724 STM32 Nucleo-64 boards user manual. https://www.st.com/resource/en/user_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf

STMicroelectronics (05-2022) RM0091 reference manual: STM32F0x1/ STM32F0x2/ STM32F0x8 advanced ARM-based 32-bit MCUs. https://www.st.com/resource/en/reference_manual/rm0091-stm32f0x1stm32f0x2stm32f0x8-advanced-armbased-32bit-mcus-stmicroelectronics.pdf, RM0091-Rev.10

STMicroelectronics (12-2022) Using the STM32F0/F1/F3/Cx/Gx/Lx series DMA controller. https://www.st.com/resource/en/application_note/an2548-using-the-stm32f0f1f3cxgxlx-series-dma-controller-stmicroelectronics.pdf, AN2548-Rev.8

Suhendra V, Mitra T, Roychoudhury A et al (2005) Wcet centric data allocation to scratchpad memory. In: 26th IEEE international real-time systems symposium (RTSS'05), IEEE, p 10

Tan KC, Lee TH, Khor EF (2002) Evolutionary algorithms for multi-objective optimization: performance assessments and comparisons. Artif Intell Rev 17(4):251–290

Tarjan R (1973) Testing flow graph reducibility. In: ACM symposium on theory of computing, pp 96–107

TeamPlay Consortium (2020) Deliverable D4.5—report on energy usage analysis and on prototype—version 1.0

Udayakumaran S, Barua R (2003) 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, pp 276–286

Verma M, Marwedel P (2007) Scratchpad overlay approaches for main/scratchpad memory hierarchy. In: advanced memory optimization techniques for low-power embedded processors. Springer, pp 83–119

Wegener S, Nikov KK, Nunez-Yanez J et al (2023) Energyanalyzer: using static wcet analysis techniques to estimate the energy consumption of embedded applications. arXiv preprint arXiv:2305.14968

Whitham J, Audsley N (2009) Implementing time-predictable load and store operations. In: International conference on embedded software (EMSOFT), pp 265–274

Yang XS (2012) Flower pollination algorithm for global optimization. In: International conference on unconventional computing and natural computation. Springer, pp 240–249

Yang XS, Karamanoglu M, He X (2014) Flower pollination algorithm: a novel approach for multiobjective optimization. Eng Optim 46(9):1222–1237

Yang Y, Wang M, Yan H et al (2010) Dynamic scratch-pad memory management with data pipelining for embedded systems. Concurr Comput: Pract Exp 22(13):1874–1892
Zitzler E (1999) Evolutionary algorithms for multiobjective optimization: methods and applications, vol 63. Citeseer
Zitzler E, Thiele L (1999) Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. IEEE Trans Evolut Comput 3(4):257–271

Shashank Jadhav is currently a doctoral candidate at the Institute of Embedded Systems, Hamburg University of Technology. He received a joint master's degree in Mathematical Modelling in Engineering from the University of L'Aquila, Italy and the University of Hamburg, Germany. His research interests are in compiler-level multi-objective optimizations, with a particular focus on memory allocation-based code optimization techniques for real-time systems.

Heiko Falk received the PhD degree in computer science from the University of Dortmund, Germany, in 2004. From 2004 until 2011, he worked as assistant professor in the embedded systems group at the Technical University of Dortmund. Between 2011 and 2015, he worked as full professor for embedded systems and real-time systems at Ulm University (Germany). Since 2015, he is full professor at Hamburg University of Technology (TUHH) and heads the Institute of Embedded Systems. His PhD focused on high-level source code optimizations. Typical embedded multimedia applications only use a small fraction of their execution time to compute audio or video data. Most of the execution time is used to evaluate complex control flow. Motivated by this observation, he developed novel techniques for control flow optimization at the source code level. Another focus of his work is on code generation and optimization for performance and predictability of safety-critical real-time systems. The WCC compiler initially established by him and developed by the research teams led by him is the currently only known compiler which is able to systematically reduce the worst-case execution time (WCET) of programs by tightly integrating static timing analyses into the code generation and optimization stage. He works on novel concepts to handle parallelism during real-time analysis and optimization. Mutual interferences between tasks running preemptively on the same CPU, or between tasks running on different cores and using shared resources are the key challenges of his current work. Since embedded systems regularly have to meet various additional design constraints besides real-timeliness, his current research also puts an emphasis on multi-criterial optimizations. He and his team develop novel compiler optimizations that are able to systematically trade, e.g., performance versus energy efficiency versus code size. He regularly publishes his research results at prestigious conferences and journals edited, e.g., by Springer, ACM, or IEEE. He regularly serves as program committee member and reviewer for the international research community, and he was conference chair and local host of the 2023 edition of the ACM/IEEE Embedded Systems Week (ESWEEK).