Worst Case Execution Time Oriented Code Optimization of Hard Real-Time Multicore Systems

Vom Promotionsausschuss der
Technischen Universität Hamburg

zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation

von
Dominic Oehlert, M.Sc.

aus
Hameln

2021

Gutachter:	Prof. Dr. Heiko Falk Prof. Dr. Sibylle Schupp
Tag der mündlichen Prüfung:	23. September 2021

Zusammenfassung

Eine Vielzahl von modernen computergestützten Systemen unterliegen zeitlichen Beschränkungen welche eingehalten werden müssen. Die einzuhaltenden zeitlichen Schranken ergeben sich typischerweise aus den physikalischen Prozessen, in welche das System eingebunden ist. Da diese Systeme ihre Berechnungen innerhalb dieser gegeben Zeitschranken bewältigen müssen, werden sie *Echtzeitsysteme* genannt. Falls das Nichteinhalten einer solchen Zeitschranke die Qualität des Ergebnisses nicht nur verringert, sondern komplett unbrauchbar macht, wird von einem *harten* Echtzeitsystem gesprochen. Mittels gegebener Analysemethoden kann überprüft werden, ob ein hartes Echtzeitsystem jemals seine gegebenen Zeitschranken überschreiten kann. Falls ein hartes Echtzeitsystem seine zeitlichen Schranken unter Umständen nicht einhalten kann, sind Optimierungen in der Software und/oder in der Hardware nötig.

Durch die starke Verbreitung im Anwenderbereich haben Multi-Core-Architekturen in den letzten Jahren auch ihren Weg in den Bereich von harten Echtzeitsystemen geschafft. Aufgrund geteilter Ressourcen in einer Multi-Core-Architektur ergeben sich zeitliche Einflüsse, welche bei der Analyse von diesen neuen Echtzeitsystemen beachtet werden müssen. Gleichzeitig ergeben sich hierdurch auch neue Ansatzpunkte für Multi-Core-spezifische Optimierungen, um die Einhaltung von den zeitlichen Schranken des Systems zu garantieren.

Diese Arbeit präsentiert neue Compiler-basierte Optimierungsansätze auf unterschiedlichen Abstraktionsebenen für Multi-Core-Systeme, um deren Echtzeitfähigkeit zu verbessern. Hierfür werden zwei Speicherallokationsverfahren für unterschiedliche Busarbitrierungsstrategien in Multi-Core-System vorgestellt, welche explizit die Eigenschaften des jeweiligen Arbitrierungsverfahren ausnutzen. Um Optimierungen und Analysen zwischen einer gut skalierenden, jedoch sehr abstrakten Systemebene und einer detaillierten, jedoch schlecht skalierenden mikroarchitekturellen Ebene zu ermöglichen, wird außerdem eine auf ganzzahlig linearer Programmierung basierende Ableitung von Metriken für Zugriffe auf geteilte Ressourcen vorgestellt. Diese ermöglicht unter Anderem sowohl effiziente Analysen, als auch Optimierungen für Multi-Core-Systeme mit einer sogenannten „work-conserving“ Busarbitrierungsstrategie. Des Weiteren wird ein neuartiges Traffic-Shaping-Verfahren auf Assembler-Code-Ebene vorgestellt, welches das Zugriffsverhalten eines gegebenen Programms automatisiert umformen kann. Hiermit kann das Prinzip des Traffic-Shapings zur Verbesserung der Echtzeitfähigkeit auf Multi-Core-Architekturen übertragen werden, in welchen es typischerweise keine Möglichkeit einer präzise gesteuerten Zugangskontrolle von geteilten Ressourcen gibt.

Die Evaluationen der vorgestellten Optimierungsmethoden zeigen, dass die explizite Beachtung von Buseigenschaften innerhalb der Optimierungen einen klaren Vorteil erbringen kann. Verglichen mit Optimierungen, welche die Buseigenschaften nicht explizit beachten, können mehr Systeme nach den vorgestellten Optimierungen alle geforderten Zeitschranken einhalten.

Kurzzusammenfassung

Diese Arbeit präsentiert neue Compiler-basierte Optimierungsansätze auf unterschiedlichen Abstraktionsebenen für Multi-Core-Systeme, um deren Echtzeitfähigkeit zu verbessern. Zur Erhöhung der Skalierbarkeit von Optimierungen und Analysen mit Einblick auf die mikroarchitekturielle Ebene wird außerdem eine auf ILP basierende Ableitung von Metriken für Zugriffe auf geteilte Ressourcen vorgestellt. Die Evaluationsergebnisse zeigen, dass die präsentierten Optimierungen in vielen Fällen die Echtzeitfähigkeit von Multi-Core-Systemen signifikant verbessern können.

Abstract

This thesis introduces new compiler-based optimization approaches for multi-core systems to improve their hard real-time characteristics on varying levels of abstraction. To increase scalability of optimizations and analyses with an insight into the microarchitectural level, an ILP-based derivation of shared resource access metrics is introduced. The evaluation results show, that in many cases, the optimization approaches introduced in this thesis can significantly improve the hard real-time characteristics of multi-core systems.

Publications

Parts of this thesis have been published in a peer-reviewed scientific journal, proceedings of conferences and workshops, a book chapter or as technical report. These publications are listed in a chronological order below. The precise contribution of each listed paper to this thesis is further discussed in the respective chapter.

- Dominic Oehlert, Arno Luppold and Heiko Falk. "Practical Challenges of ILP-based SPM Allocation Optimizations". In: *Proceedings of the 19th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, St. Goar / Germany, May 2016, pp. 86-89. DOI: 10.1145/2906363.2906371.

- Dominic Oehlert, Arno Luppold and Heiko Falk. "Bus-aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems". In: *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS)*, Dubrovnik / Croatia, June 2017, pp. 1:1-1:22. DOI: 10.4230/LIPIcs.ECRTS.2017.1

- Dominic Oehlert and Heiko Falk. "WCET Analysis of Automotive Buses using WCC". In: *Proceedings of the DATE Workshop on New Platforms for Future Cars, Dresden / Germany, March 2018*.

- Dominic Oehlert, Arno Luppold and Heiko Falk. "Mitigating Data Cache Aging through Compiler-Driven Memory Allocation". In: *Proceedings of the 21st International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, St. Goar / Germany, May 2018, pp. 58-61. DOI: 10.1145/3207719.3207731

- Dominic Oehlert, Selma Saidi and Heiko Falk. "Compiler-Based Extraction of Event Arrival Functions for Real-Time Systems Analysis". In: *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS*), Barcelona / Spain, July 2018, pp. 4:1-4:22. DOI: 10.4230/LIPIcs.ECRTS.2018.4

- Dominic Oehlert, Arno Luppold and Heiko Falk. "Compilation for Real-Time Systems - An Overview of the WCET-Aware C Compiler WCC". In: *Proceedings of the 9th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS)*, Barcelona / Spain, July 2018. DOI: 10.15480/882.2271

- Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert and Arno Luppold. "Automated generation of time-predictable executables on multi-core". In: *Proceedings of the 26th International Conference on Real-Time Networks and Systems (RTNS)*, Poitiers / France, October 2018, pp. 104-113. DOI: 10.1145/3273905.3273907

- Arno Luppold, Dominic Oehlert and Heiko Falk. "Evaluating the Performance of Solvers for Integer-Linear Programming". Technical Report. Hamburg / Germany: Hamburg University of Technology, Institute of Embedded Systems, November 2018. DOI: 10.15480/882.1839

- Dominic Oehlert, Arno Luppold and Heiko Falk. “Favorable Adjustment of Periods for Reduced Hyperperiods in Real-Time Systems”. In: *Proceedings of the 22nd International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, St. Goar / Germany, May 2019, pp. 82-85. DOI: 10.1145/3323439.3323975
- Dominic Oehlert, Selma Saidi and Heiko Falk. “Code-Inherent Traffic Shaping for Hard Real-Time Systems”. In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*, New York City / USA, October 2019. DOI: 10.1145/3358215
- Arno Luppold, Dominic Oehlert and Heiko Falk. “Compiling for the Worst Case: Memory Allocation for Multi-task and Multi-core Hard Real-time Systems”. In: ACM Transactions on Embedded Computing Systems (TECS), Vol. 19, No. 2, ACM, March 2020. DOI: 10.1145/3381752
- Heiko Falk, Shashank Jadhav, Arno Luppold, Kateryna Muts, Dominic Oehlert, Nina Piontek and Mikko Roth. “Compilation for Real-Time Systems a Decade After PREDATOR”. In: A Journey of Embedded and Cyber-Physical Systems, pp. 151-169, Springer, August 2020. DOI: 10.1007/978-3-030-47487-4_10
- Dominic Oehlert, Edward Umaña Williams and Heiko Falk. “Work-In-Progress: Fine-Grained On-Chip Energy Measurement of a Real-Time Multi-Core Processor”. In: *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, December 2020, pp. 383-386. DOI: 10.1109/RTSS49844.2020.00044

Contents

1 Introduction

Computers have become ubiquitous in the modern day environment. They entered virtually all possible areas, such that nearly every aspect of our everyday life is influenced by a calculation made by a computer along the way. That involves more obvious ones, such as which bus to take according to the smartphone application of the local transport company, and less obvious ones, such as the amount of fuel to be injected dependent on the gas pedal position of a car. This immense immersion of computers into different domains has been made possible by their constant improvement in terms of performance, size, efficiency and costs. While they were once huge machines, weighing tons and requiring several thousands of Watt to operate while barely computing a single operation per second, they transformed into multi-core architectures with a throughput of more than a billion operations per second while being barely larger than a fingernail.

These modern architectures have also entered highly timing critical domains, which turns them into so-called *hard real-time* systems. Hard real-time systems are defined by their requirement to calculate the correct output to a given set of input signals in a given time interval. If the correct solution is not ready by the required deadline, the complete system may fail. Using timing analyses, it can be reasoned if such a deadline violation can ever occur or if it is guaranteed that the system will never miss its deadline. In case a given hard real-time system may violate a deadline, it has to be considered broken and requires a fix before it can be used. This fix can be in terms of hardware or software – or a combination of both. In terms of hardware, this could be done by using, e.g., a more powerful processor. Yet, switching the hardware may be difficult or even impossible due to several reasons:

Costs The more powerful architecture will most likely cost more than the previous one, hence will increase the overall costs of the system. Additionally, using a new architecture may also lead to secondary costs, such as porting existing software or repeating certain analyses (e.g., power measurements).

Safety For many domains of hard real-time systems, a safety certification of the hardware and its corresponding software is required (e.g., as required by ISO 26262 [Int18b] in the automotive domain or DO-178C [RTC12] in the avionics domain). A change of hardware would most likely need a re-certification, of which the outcome may be uncertain.

Legacy The system may include legacy code which may be difficult to port to a new platform. Beside legacy code, the old system may include legacy interconnection standards which newer architectures do not support anymore.

While optimizations on the software side may require a safety re-certification of the system as well, it does not increase the costs due to new hardware or involve the problems of porting legacy code to a new platform. Optimizations on the software

side therefore open up vast opportunities to reach the required deadlines which the system violated beforehand.

In order to optimize the software of the deadline-violating system, the system designer can try to optimize the source code of the programs manually. Beside being cumbersome and potentially error-prone, the means of this may be limited depending on the programming language used. As the programmer is only able to optimize the source code manually, whereas all decisions close to the hardware are chosen during the compilation process by the compiler, the programmer has no direct influence on these decisions. Additionally, the manual optimizations would have to be carried out iteratively in a loop, each time performing a timing analysis after a modification, ensuring that the fix actually is improving the worst-case timing. Due to this, utilizing worst-case-oriented compiler optimizations offers a great chance to overcome these burdensome obstacles and to automatically improve the worst-case timing during the compilation process.

The significant characteristic of a multi-core architecture lies in its parallel cores and therefore its capability to execute multiple programs in a truly parallel fashion. While this is highly beneficial from a performance-view, it also highly increases the complexity to estimate tight upper bounds on the execution time of a program running on such an architecture. With multi-core architectures entering the domain of hard real-time systems, new challenges for worst-case-oriented optimizations arise. As the influence of interconnections on the execution timing in a multi-core architecture can be one of the most critical spots, it therefore also opens up whole new opportunities to optimize this specific aspect. The overall ambition of such multi-core-aware optimizations is to use the shared resources as efficient as possible in order to improve worst-case timings. The actual means by how this is achieved can be manifold: allocating the right parts of a program to a private resource, adjusting the access pattern to a shared resource, adapting the arbitration scheme, improving the task-to-core allocation, etc.

In order to automatically optimize multi-core-based hard real-time systems by employing optimizations on a compiler-level, new approaches are explored in the following chapters. Therefore, two memory allocation techniques are presented which explicitly take bus-related timings into account to generate allocations which are specifically tailored towards the multi-core system and its behavior. In order to analyze and optimize a hard real-time multi-core system efficiently on a compiler-level, a method to derive an abstract metric to represent a program's shared resource behavior is presented. This abstract metric is used as a middle ground between a system-level approach, where each program of a system is considered as a black box and is only characterized by a few values, and a deeply integrated approach, where the influence of each instruction is precisely depending on the parallel cores. While the latter one does not scale well, the system-level approach heavily reduces the potential of code-based optimizations. By deriving an abstract metric representing a program's shared resource behavior from the code-level, it enables a larger potential for compiler-based optimizations while improving the scalability. This is used for one of the proposed memory allocation techniques, as well as for a novel traffic shaping-based optimization. The proposed traffic shaping-based optimization takes the general idea of traffic shapers and applies it to a standard multi-core system by

only adapting the code to enforce a given traffic profile, thereby improving worst-case timings.

1.1 Contribution

This thesis proposes novel methods to optimize multi-core-based hard real-time systems on a code-level by explicitly capturing multi-core-related characteristics. Additionally, a precise formulation for deriving an abstract model to describe the shared resource access behavior of a task on a code-level is presented. The following list details the individual contributions of this thesis and their relation to each other:

- An ILP-based worst-case timing-oriented static program memory allocation for multi-core systems with a Time Division Multiple Access (TDMA)-arbitrated bus. The proposed ILP formulation is able to precisely predict potential bus-states on a CPU cycle-level and thereby to estimate bus-related timings dependent on the allocation.

- A formalized description of deriving so-called *event arrival functions* from a given task on a code-level using ILP. By using event arrival functions as an abstract metric to describe shared resource access behaviors and by deriving them directly from a code-level, the scalability of system-level approaches and the deep insight of low-level analyses can be combined.

- A method to include traffic shaping behavior into a task on a code-level to improve worst-case timing characteristics in a multi-core system. The proposed method enforces an arbitrary traffic profile function of a task only by altering the code of a task, not relying on OS- or hardware-specific features. Therefore, a genetic algorithm-based approach as well a greedy heuristic are presented. Both approaches rely on the derivation of event arrival functions on code-level, constituting a practical application.

- A static memory allocation focusing on optimizing the overall real-time capability of a multi-core system. This novel approach mainly focuses on multi-core systems with a work-conserving bus arbitration, as work-conserving bus arbitration policies are the most common in today's bus-based architectures. By selecting program parts and data objects not only to improve the timing of a single core, but also of concurrent cores, the approach tries to optimize the system as a whole and achieve overall schedulability. An ILP-based approach as well as a heuristic are presented, both relying on event arrival functions derived on a code-level.

- In order to evaluate and show the actual applicability and practicality, all proposed algorithms were implemented in the WCET-aware C Compiler (WCC) and tested for ARM-based architectures. Additionally, a case study set in the automotive domain demonstrates the applicability on real-world benchmarks of the aforementioned contributions and how they can be combined.

1.2 Structure

This thesis is structured as follows: The following Chapter 2 gives an introduction into the field of multi-core architectures in hard real-time systems. Here, a general overview of multi-core architectures and their specific characteristics is given, as well as an insight into analyzing them in terms of worst-case execution time. This chapter also outlines general assumptions on the multi-core architectures used for the upcoming optimizations. Chapter 3 details the structure and features of the compiler framework in which all proposed algorithms have been implemented.

Chapter 4 introduces a static program memory allocation for multi-core architectures with a TDMA-arbitrated bus. Since TDMA enforces a complete isolation of interferences between the cores, its use in hard real-time systems is recommended. Yet, unfortunately, many multi-core Commercial Off-The-Shelf (COTS) still do not support TDMA-based bus arbitration but rather work-conserving arbitration schemes. Optimizations for such work-conserving bus arbitration schemes require a different approach, as modeling their interferences between cores on a similar low level is not suitable. To enable optimizations and analyses between a very detailed, yet poorly scaling microarchitectural level and a well-scaling, yet very abstract system-level, the derivation of event arrival functions on a code-level is presented in Chapter 5. As abstract metrics of resource demands directly derived from the code-level, these event arrival functions can be used to perform optimizations and analyses on a higher level of abstraction, while still including details from a low level. This is subsequently shown with two different optimizations supporting more complex to analyze bus arbitration schemes. Here, Chapter 6 introduces the idea of *code-inherent* traffic shaping in hard real-time systems and how it can be applied automatically. Subsequently, Chapter 7 presents a memory allocation approach for program parts and data objects to achieve an overall schedulability, especially focused on multi-core architectures with a work-conserving bus arbitration. Each of the chapters described in this paragraph discusses relevant related work in their field and evaluates the proposed approaches.

The introduced optimizations and analyses are used in a case study in Chapter 8 to fix an initially "broken" real-time system set in the automotive domain. It is shown how the presented approaches can improve the real-time characteristics of a specific multi-core system and how they can be combined.

Finally, this thesis closes with Chapter 9, drawing overall conclusions and highlighting possible future work.

2 Real-Time Multi-Core Systems

Real-time systems have become ubiquitous in nearly every single aspect of everyday life over the course of the last decades. They can be found in the more obvious domains, e.g., in automotive or avionics, but also in the lesser thought-of ones, like, e.g., in washing machines or televisions. Real-time systems are characterized by the fact that they are not only required to compute functionally correct results to a given program, but that the computation also has to be finished in a specified time frame. The consequence of a belated arrival of a computation result is often seen to be as critical as returning a wrong result or no result at all. Further on, systems can be categorized into so-called *hard* or *soft* real-time systems. Loosely based on the definitions by Manacher [Man67], soft- and hard real-time systems can be defined as follows:

Definition 2.1 (Soft Real-Time System)**.** *A soft real-time system needs to perform a sequence of calculations in a defined time frame, whereas minor timing violations are acceptable, yet not desirable.*

Definition 2.2 (Hard Real-Time System)**.** *A hard real-time system needs to perform a sequence of calculations in a defined time frame, whereas any timing violation results in an unusable result and is not tolerable.*

A timing violation of a hard-real time system can result in a complete system failure and even endanger human lives. These rigid time frames, within which it is required to return the correct result, typically stem from surrounding environment the system is placed in. In case of, e.g., an electronic combustion engine control, the physical laws describing the motions of the cylinders and crankshaft dictate timing requirements. If the result comes in too late, it becomes unusable as the surrounding required it earlier.

Real-time systems are typically part of larger systems, which places them in the group of so-called *embedded systems*. Based on the definition by Marwedel [Mar11], an embedded system is defined as follows:

Definition 2.3 (Embedded System)**.** *An embedded system processes incoming information and is embedded into a larger context or product.*

Driven by the ever-growing demand on computing, the complexity and with it the performance of processing units have been continuously growing since the beginning of early microprocessors in the 1970s. Led by predictions such as the well-known “Moore’s law” [Moo65], the number of transistors per area exponentially grew, whereas the operating frequency could be increased due to the smaller fabrication dimensions, leading to a steady increase in performance. On the downside, shrinking transistor sizes, which leads to increased leakage currents, and a higher operation frequency contribute to a larger power dissipation [Bor07]. This resulted in hitting the so-called “power wall”: A higher performance could not be reached by increasing the operation frequency as the resulting heat becomes too large to dissipate. As an

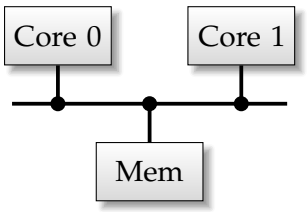


Figure 2.1. – An exemplary single bus interconnection architecture.

opposing solution, multiple processors are integrated on the same chip, becoming so-called *multi-cores*.

While multi-cores open great opportunities in the domain of hard real-time systems, they also introduce new challenges. The access latency of a core to a shared resource, such as a bus or a memory, may depend on the behavior of the other cores in the multi-core system. Hence, the execution time (and thus the worst-case execution time as well) may not only depend on the program itself, but also on the programs running concurrently on the other cores.

In the following, fundamental architectural concepts of multi-core systems and the timing analyses of such are introduced.

2.1 Architecture Specifics

Multi-core systems greatly vary in their structure and design choices. The following section will give a brief overview of different key aspects of a multi-core system and their prominent implementations. As this section is only intended to draw an outline of the wide area of multi-core architectures, the interested reader is referred to further reading [Moy13, PH13].

2.1.1 Interconnections

One of the major architecture features of a multi-core system is the type of used interconnection network. Whereas in a traditional single-core system, only one core has to be connected to potentially multiple on-chip resources, multiple cores need to be connected to the resources in a multi-core system. Interconnection networks differ in the degree of parallelism, area and power requirements, predictability and fairness. In the following, single and hierarchical buses, bus crossbars and networks-on-chip are discussed.

Single and Hierarchical Buses

The single bus interconnection network is the simplest type. It connects all resources using only a single shared bus. Hence, its area requirement and power dissipation is very low. Figure 2.1 depicts such an exemplary single bus interconnection architecture with two bus masters and a shared memory. Whereas its design pays off in terms of simplicity, it does not scale well with a larger number of cores or, in general, with the number of attached participants. With a larger number of bus masters, the bus contention typically increases, hence transforming the bus into the bottleneck of the system. Simultaneously, the bus frequency is bound to the slowest bus participant.

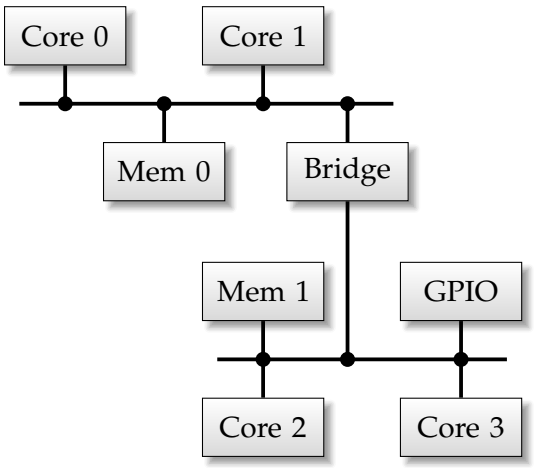


Figure 2.2. – An exemplary hierarchical bus interconnection architecture.

While computing cores and memories may use a higher frequency, thus increasing the overall performance, additional slower clocked participants, e.g., UART or GPIO controllers, force the bus frequency to be lower. An exemplary family of System-on-a-Chip (SoC) systems using such a single bus interconnection are the DaVinci processor-based SoCs from Texas Instruments [Tal07].

A subset of the just named drawbacks can be eliminated or reduced by the introduction of a hierarchical bus architecture. An exemplary system with a hierarchical bus is depicted in Figure 2.2. Several single buses can be connected by bus bridges which can typically transfer information in both directions. One main advantage of a hierarchical bus is the ability to use different bus frequencies. High performance computing cores and their indispensable bus slaves can be placed onto one high frequency bus, whereas low frequency peripherals can be placed onto a second low frequency bus. An exemplary multi-core system with a hierarchical bus is the LEON4-based GR740 [HAW+15] quad-core processor from Cobham Gaisler. While this potentially raises the bus frequency and enhances the energy efficiency of the system, the bus still easily becomes the bottleneck when the number of bus masters is increased.

Crossbars and Network-on-Chips

To overcome bus contention-based bottlenecks in a multi-core system, the available communication bandwidth can be increased. So-called crossbars are one possibility to do so. An exemplary architecture with a crossbar is shown in Figure 2.3. A crossbar enables multiple transfers to happen in parallel. Each bus slave can be connected to each bus master directly over a so-called interconnection matrix. In case multiple masters try to access the same slave simultaneously, an arbiter logic has to decide which connection is established. One example architecture typically found in the hard real-time domain utilizing a crossbar interconnection is the TriCore AURIX family [DMK+18]. Whereas a crossbar architecture can greatly improve the performance of a multi-core system due to lower contention, its required area scales poorly (quadratic with the number of participants [Ben06]).

A so-called Network-on-Chip (NoC) interconnection architecture achieves a better scalability by transferring the structure of large-scale computer networks onto a chip-

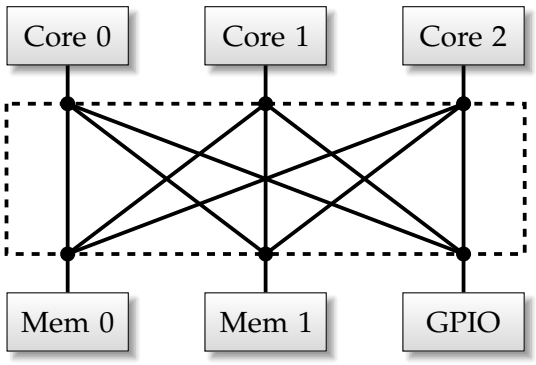


Figure 2.3. – An exemplary crossbar architecture.

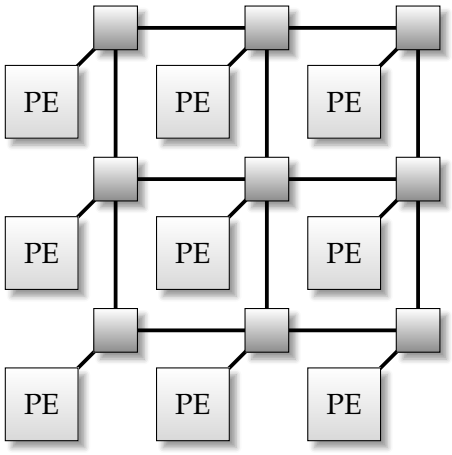


Figure 2.4. – An exemplary Network-on-Chip architecture.

level. An exemplary architecture is given in Figure 2.4. In contrast to the previously discussed architectures, the communication inside a NoC is based on single packets to be sent across the network. Each Processing Element (PE) (such as, e.g., a single core) is connected to a router. These routers are represented in Figure 2.4 as the square to which a processing element is connected. The number of elements per router is depending on the network architecture, as well as the routing algorithms (which way a packet can take across the network) and the flow control (how and when does a router accept a packet). An exemplary commercial architecture which incorporates a Network-on-Chip is the ARM Cortex A9-based SPEAr1340 processor from STMicroelectronics [STM12].

2.1.2 Memories

Beside the previously discussed interconnection network of a multi-core system, the architecture of the used memories is crucial to the system's performance and analyzability. Memories build the foundation of any processing system, as without any memory, no programs and no data can be hold to be executed and used by a processor. In a multi-core system, the memory architecture features different aspects which are partially orthogonal. In the following, a brief overview of the key aspects

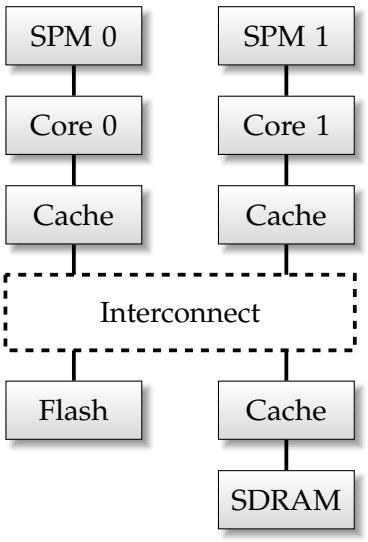


Figure 2.5. – An exemplary multi-core architecture with different types of memory.

of the memory architecture inside multi-core systems is given. For this, an exemplary multi-core architecture with several memory elements is shown in Figure 2.5.

Architectures may use different memory technologies, often in a hierarchical fashion, to exploit the specific features of a memory. Figure 2.5 depicts four different types of memory commonly used in modern multi-core architectures.

Synchronous Dynamic Random Access Memory (SDRAM). An SDRAM is, as already denoted by the name, a *dynamic* memory, which means that its content has to be refreshed periodically (which is typically automatically done by an integrated controller), as it otherwise loses the contained information. It offers large capacities in the magnitude of Gigabytes. Its access latency is in the range of dozens of nano seconds. In terms of timing analyzability, SDRAMs feature a rather complicated deterministic behavior. Modern SDRAMs follow a so-called *distributed refresh* strategy. Here, not the entire memory is refreshed at once, but every row is refreshed periodically, whereas the row refreshes are spaced equidistant in time. If an access happens during a refresh, the handling of this access is delayed by the SDRAM until after the refresh of the row has finished. The exact refresh frequency and the duration of a single refresh depend on the actual SDRAM. A typical value for the refresh period is 7.8 µs, whereas the refresh cycle of a modern DDR3 SDRAM with 1 Gbit takes around 110 ns [Int18a]. Additionally, even when neglecting the potential additional latency of an access due to a concurrent refresh, the access latency to a specific word in the memory may vary greatly. This is due to caching effects inside the SDRAM, where previously accessed rows are buffered in a so-called *row buffer* for a certain amount of time [WPG16]. While the potentially large and dense memory of an SDRAM is interesting for embedded systems, its timing behavior is difficult to predict precisely which lowers its attractiveness for hard real-time systems.

Flash. Flash memory is a static and non-volatile memory type often used as a main memory for many common microcontrollers. As it is non-volatile, Flash memories

can be used to store the actual program(s) to be executed, which may be moved to other, volatile memories during startup. Its typical memory capacity ranges from a few hundred kilobytes up to several gigabytes. A single random read access is typically in the range of 50 ns to 100 ns, whereas subsequent accesses to addresses in the same range require significantly less time [Cyp18, Mic13]. In contrast to SDRAM, this timing behavior of burst-accesses can be very well predicted. As the number of Flash write cycles are limited (in the range of 100 000) and writing to the Flash takes significantly longer (magnitude of tens to hundreds of microseconds), Flash memory is typically only used for instructions (assuming non-self-modifying code) and read-only data.

Scratchpad Memory (SPM). A so-called scratchpad memory is a fast Static Random Access Memory (SRAM) directly mapped into the address space. This memory is typically tightly coupled with a processing core, enabling random accesses (read or write) in a single CPU cycle, independent from the clock frequency. While this enables the highest potential performance and analyzability, its comparably less-dense structure restricts its size to a typical range of few dozen kilobytes [NXP09b, Inf09a, Inf14]. As it is a volatile memory, the content needs to be loaded into it before it can be used. This can be either done during the startup of the system, or during the execution if its content is not static. Due to their low latency and very deterministic behavior, SPMs are widely used in the domain of hard real-time systems.

Caches. Caches serve as small, yet fast buffer memories for larger, yet slower main memories. As can be seen in Figure 2.5, the slower memory might be another cache as well, allowing so-called hierarchical cache structures. A cache typically consists of SRAM with additional logic. The additional logic decides which subset of buffered memory content should be replaced in favor of new content to be loaded into the cache. In case the cache receives a memory access request for memory content which it currently stores, this access is called a *hit* and can be handled within very low latency (a tightly coupled cache typically has an access time of a single cylce in case of a cache hit). Otherwise, the access is labeled a so-called *miss* and is handled by regularly accessing the slower memory. As the content of the cache is depending on the current system's state and its history, it is naturally less deterministic and therefore harder to analyze in terms of worst-case execution time when compared to, e.g., SPMs. Furthermore, the analyzability heavily depends on the cache's so-called replacement policy, deciding which cache content to be replaced in case of a previous cache miss [Rei08].

Beside the actual, potentially different memory technologies used, the memory architecture of a multi-core system is also characterized by the uniformity of access latencies in regard to the different cores. If the memory access latencies inside a system are independent from the cores which initiate the accesses, the memory architecture is classified as Uniform Memory Access (UMA). In contrast, so-called Non-Uniform Memory Access (NUMA) memory architectures feature access latencies which may depend on the core which is initiating the access, as some memories are coupled directly to processing cores. The architecture in Figure 2.5 displays a NUMA system, as each core has access to its own private memory. Additionally, Figure 2.5 features

shared memories which are attached directly to the interconnection network, offering equal access latency to all cores. Whether a processing core may access another core's local memory (e.g., core 0 accessing SPM 1) or only the corresponding core can access its local memory is depending on the architecture. A NUMA-based architecture with local memories private to each corresponding core can increase the system's predictability, as accesses to the local memory result in a fixed latency due to the fact that no other processing core may access it concurrently.

2.1.3 Arbitration Schemes

While an interconnection network enables the access to shared resources or communication, it is left open to an arbitration scheme to decide how the shared interconnection network can be accessed to avoid unwanted simultaneous accesses. In the following, different arbitration schemes for bus-based interconnection networks are discussed. These can be applied to simple single bus interconnection architectures, hierarchical bus architectures (here each bus hierarchy can have its own arbitration scheme) or also crossbars (effectively each shared resource can have its own arbitration scheme). The arbitration in NoCs works slightly different, as network principles like routing or flow control are the key factors here. Discussing these principles would go beyond the scope of this thesis and is therefore left out, the interested reader is referred to further reading [Ben06].

In case a multi-core architecture features a bus-based interconnection network, this bus is shared by several participants. When multiple participants try to access the shared bus at the same time, these accesses can be divided by using Time Division Multiplexing (TDM), Code Division Multiplexing (CDM) or Frequency Division Multiplexing (FDM). Although there are examples of using CDM or FDM for on-chip bus architectures [NSD09, Stó12], these applications are currently still restricted to the domain of research and are not present in any modern multi-core architectures. As only one participant can access the shared bus at a time with TDM, an arbiter has to decide which participant can get access to the bus at which point in time, such that accesses do not collide. To solve this issue, a variety of bus arbitration schemes have been developed, differing in fairness among the participants and their average-case, as well as their worst-case performance. Independent from the arbitration scheme, the bus arbitration is typically done in a non-preemptive manner, meaning if a bus access has been granted by the bus arbiter to a core, this access cannot be preempted by another core until it is finished [SJ96]. This is done to avoid additional buffers or complex logic to support aborted bus accesses. In the following, three well-known bus arbitration schemes are presented.

Fixed Priorities As the name suggests, each core has a fixed priority in case of an arbitration following a fixed priorities scheme. If multiple bus masters try to access the shared bus at the same time, the master with the highest priority will always win the arbitration. Therefore, a fixed priorities arbitration is a so-called *unfair* arbitration [Dal04], as a high priority master could continuously take control over the bus, while all other masters would starve. This also relates to a comparably poor analyzability for multi-core hard real-time systems, as it is impossible to derive a worst-case timing for a single access without analyzing the behavior of all higher

priority cores. It is a *work-conserving* arbitration, as the bus is never idle in case there is at least one master requesting the bus.

Round Robin (Fair) This arbitration scheme rotates the masters' bus priorities after each performed bus access. Thus, a bus master requesting a bus grant will receive it after at most $N_C - 1$ bus accesses of other masters (with N_C being the number of masters of the bus) at any time. This makes round robin an arbitration with *strong fairness*. Due to this, multi-core systems with a round robin-based bus arbitration are better analyzable in terms of worst-case behavior than with a fixed priority one, as each access duration is bounded by a tight upper limit. Using a more detailed analysis of the other cores' behavior, this worst-case can also often be further reduced for some accesses [Kel15, JHH15]. Since it is also a work-conserving arbitration, it also features a good average-case performance.

Time Divison Multiple Access (TDMA) In case of a TDMA arbitration, the arbiter follows a so-called *bus schedule* cyclically. The bus schedule consists of different *time slots*, whereas each slot has a defined duration and a corresponding bus master. During each slot, only the associated master can access the bus, whereas each other master's bus access is denied by the arbiter. The bus arbiter also has to ensure that the full access still fits inside the current bus slot, such that slots are not overlapping due to, e.g., memory latency. Therefore, each bus slot has a certain amount of so-called grant cycles, where every bus access from the corresponding bus master will be granted. In case the bus master corresponding to a slot does not try to access the bus during the slot's duration, the bus is idling during this time, independent whether other masters try to access the bus or not. Depending on the exact layout of the bus schedule, TDMA may have a *weak fairness* or even a *strong fairness* [Dal04]. The very static behavior makes TDMA an ideal arbitration for analyzing worst-case timings inside a multi-core system, as it has very tight upper limits on each access' worst-case duration. Furthermore, it enables a complete timing isolation between the bus masters, as the access behavior of one master cannot influence the bus-related timings of any other master in the system. Therefore the use of TDMA in multi-core hard real-time systems is being advocated [CFG+10]. Yet, due its strict behavior, the average-case performance of TDMA is considerably worse than, e.g., compared to a round robin arbitration, as the bus may have large idling times despite some bus masters are trying to access it. This makes TDMA also a *non work-conserving* bus arbitration.

2.1.4 Exemplary Base Architecture

This section introduces an exemplary multi-core architecture which is used as a base for optimizations in the following chapters. This base architecture is depicted in Figure 2.6 with four parallel cores, although the actual number of cores is not fixed. For later evaluation purposes, the number of cores is in the range of 2 to 8. Each core has its own private memory which is divided into a code and data section. Here, private SPMs are chosen instead of caches to increase determinism and predictability. All cores have access to shared memories which are either Flash- (for read-only parts) or SRAM-based (for read-and-write parts). The cores are connected

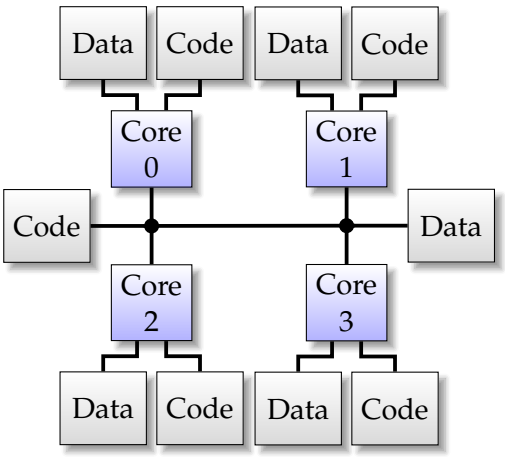


Figure 2.6. – An exemplary multi-core base architecture.

to the shared memories using a bus interconnection. A bus interconnection is chosen instead of, e.g., a Network-on-Chip due to the prevalence of bus-based architectures in current embedded multi-core systems, its lower overhead and the comparatively small number of connected cores. The bus arbitration policy is not fixed in general and is subject to the assumption of an optimization. This architecture is similar to the LEON3-based GR712RC multi-core processor [Cob20] used in the aerospace domain, but with SPMs instead of caches and a focus on the main bus hierarchy.[1]

Since this thesis focuses on the exploration of novel optimization approaches for *multi-core* architectures and their specific effects, *multi-task* systems are excluded and are part of future work, as this would go beyond the scope of this thesis. This thesis is supposed to pave the way towards optimizations and analysis for multi-task multi-core systems by closely evaluating the effects of single-task multi-core systems.

The rest of this thesis assumes that each core has a single allocated task to execute. The single tasks are not restricted to a specific activation pattern and can therefore follow any arbitrary activation patterns, provided that upper and lower bounds can be derived.

2.2 Timing Analysis

The key difference of real-time systems compared to non-real-time ones is the strict requirement to perform a certain calculation not only functionally correct, but also within a given time frame. The field of Worst-Case Execution Time (WCET) analysis arises from this requirement in order to verify if this condition is always met. The WCET of a program is the longest possible time a program may take to execute from its start to its termination. In the context of this thesis, the term WCET is used to describe this longest possible execution time *without* any interference from, e.g., concurrent cores in a multi-core system or from other tasks in a multi-task system. To formally determine the WCET of an arbitrary program is impossible in general, as it would imply to solve the so-called halting problem [Tur37] (since not only the question

[1] The private caches of the GR712RC could be used in a similar manner as SPMs as they support the locking of cache lines.

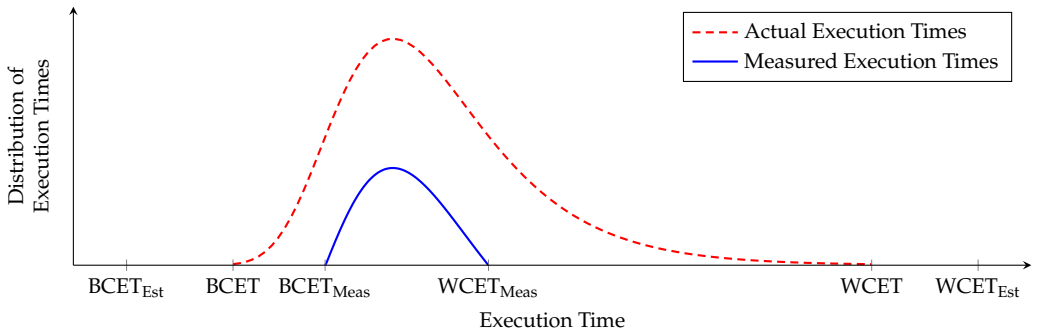


Figure 2.7. – An exemplary distribution of possible execution times of a program and its measured execution times. Adapted from [WEE+08].

if a program terminates has to be answered, but also *when*). Yet, given a formal description of the underlying processor architecture and additional information on the program to be analyzed (e.g., maximum number of loop iterations or deepest recursion depth), a safe estimation of the WCET of a program, denoted WCET_{Est} may be determined. In order to be a *safe* estimation, the following inequation has to hold:

$$\text{WCET} \leq \text{WCET}_{\text{Est}} \tag{2.1}$$

A trivial safe WCET estimation for any given program would be ∞. Therefore, a *tight* WCET estimation tries to be as close to the actual WCET as possible:

$$\text{WCET}_{\text{Est}} - \text{WCET} \rightsquigarrow 0 \tag{2.2}$$

The relationship of WCET_{Est} and WCET is displayed in Figure 2.7. The dashed curve shows the distribution of actual possible executions times of an exemplary program. Beside by its largest possible execution time, a program can also be characterized by its smallest possible execution time, named Best-Case Execution Time (BCET) (see the very left-hand side of the dashed graph). As previously mentioned, the dashed curve is typically unknown, hence also the "real" WCET and BCET of a task are unknown. The safe approximation WCET_{Est} has to be greater than or equal to the actual WCET, whereas a safe approximation of the BCET has to lower than or equal to the program's actual BCET in consequence. Figure 2.7 also shows that simple measurements of a program may result in an unsafe WCET estimation. As the WCET of a program depends on the program itself, the hardware is executed on, its initial hardware-state and the program's input, running a program for all possible configurations is typically infeasible. Therefore, only a small subset of possible configurations can be tested in a reasonable time. This leads to the worst-case execution time overall measured, named $\text{WCET}_{\text{Meas}}$. While such an estimation can be obtained easily, it does not fulfill the requirements of a safe approximation.

For the sake of readability, the term WCET will be used in the following sections and chapters as WCET_{Est}, as the actual WCET remains unknown in the vast majority of cases.

In case of a multi-core or multi-task system, the WCET itself is not safe to be used in order to prove that a system will never violate one or more of its deadlines, as it does not reflect any potential influences on the timing by other tasks or cores. The

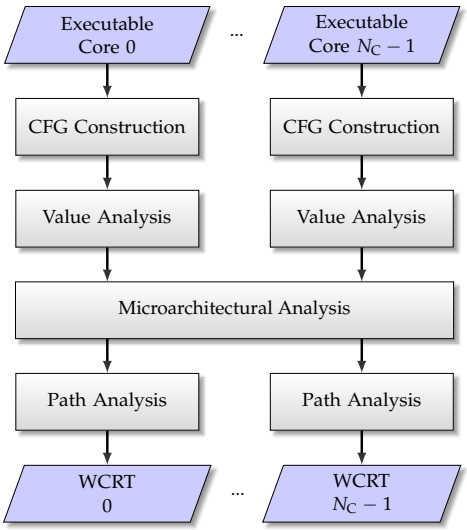


Figure 2.8. – Joint multi-core static timing analysis procedure (cf. Kelter [Kel15]).

maximum possible time between the activation of a task and its termination, *including* all potential interferences, will be denoted as the task's Worst-Case Response Time (WCRT) (similar as with the WCET_{Est}, the suffix is dropped here for the sake of a better readability). In case of a multi-task system, a task may be preempted or cannot immediately start due to higher priority tasks, leading to an increase in its WCRT. As this thesis focuses on multi-core systems with a single task allocated per core, the estimation of WCRT in a multi-task system is not discussed in a greater detail. The interested reader is referred to the thesis of Arno Luppold, which focuses on the optimization of multi-task single-core hard real-time systems on a code-level [Lup20].

In a multi-core system as depicted in Figure 2.6, each access to a shared memory is handled by the bus. Therefore, each shared memory access may be delayed for a specific amount of clock cycles until the access is actually granted due to concurrent accesses from other cores. While timing analysis of single-core systems (single- or multi-task) has been researched extensively and lead to generally accepted analysis methods, the timing analysis of multi-core architectures in real-time systems is a rather new field of research [Mar11]. Due to this, no predominant analysis techniques are yet existing [Weg17], but rather different approaches which function on different levels of abstraction. In the following, three common approaches to estimate the WCRT of a task in a multi-core system are discussed.

2.2.1 Joint Multi-Core Static Timing Analysis

A joint multi-core static timing analysis expands the principles of a "traditional" static timing analysis used for determining the worst-case execution time of a single program in a single-core system. A static timing analysis estimates the worst-case execution time of a program *safely* without actually ever executing or fully simulating it, but rather by closely inspecting its semantics with the help of *abstract interpretation* [Kil73, CC77]. As the state of a program at any given point can be described by the current content of the memory cells of the system on which it is executed, any program state can be described as the set of the system's current memory contents.

Since all modern computing systems (embedded or not) are clocked, this set describing the program's state may only change at discrete time steps. By executing an instruction and hence advancing in the execution of the program, the set describing the system's state is changed. To find the WCET of a program, all potentially existing initial system states could be tested in order to find the longest execution path. As this is typically infeasible, abstract interpretation introduces *abstract* system states. An abstract system state no longer represents one concrete set of memory cell contents, but an overapproximation of all possible states. Therefore, a static timing analyzer does not test or simulate concrete traces of a program under analysis, but analyzes an overapproximation of all possible reachable states of the system.

The typical steps of a static timing analysis are depicted in Figure 2.8, whereas the number of cores is limited to 1 for the majority of analyzers. Known commercial WCET analyzing tools employing a static timing analysis are, e.g., aiT [FH04] by AbsInt and previously Bound-T [Tid20] by Tidorum. Besides, free and open source tools such as, OTAWA [CS06], Chronos [LLMR07] or wca as a part of the platin tool kit [HHK+15] are existing as well. Whereas the previously mentioned tools only support the analysis of a single core, a comprehensive framework for determining worst-case response times in multi-core systems was presented by Davis et al. [DAI+18], yet limited to analyze single execution traces. In the following, the steps of a "traditional" static timing analysis and the additional features for a joint multi-core analysis are discussed based on the analyzer presented by Kelter [Kel15].

Control-Flow Graph Construction. At the start of the analysis, a Control-Flow Graph (CFG) of the program under analysis is created. This is typically done by inspecting the set of instructions of the program's binary executable. The created control-flow graph is a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ with a set of nodes $\mathcal{N}$ and a set of edges $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$. A single node typically represents a basic block and an edge represents the transition of the execution from one basic block to another. Following Muchnick [Muc98], a basic block is defined as follows:

Definition 2.4 (Basic Block)**.** *"A basic block is a maximal sequence of instructions that can be entered only at the first of them and exited only from the last of them." [Muc98]*

To further increase the precision of the analysis, basic blocks or even whole functions of the programs may be *virtually* cloned or inlined to represent different *execution contexts* [LM95]. *Virtually* in this case refers to the idea, that the cloning or inlining is only present in the graph representation, but the actual program remains untouched. A so-called *execution context* refers to the actual context a fragment of code can be executed in. E.g., a function can be called from different locations inside a program with different parameters, potentially leading to different execution times of the same function. Here, each location where this function is called from constitutes an execution context for this function. Similarly, basic blocks executed inside a loop may have different execution times depending on the loop iteration due to branch prediction or prefetching. Here, each loop iteration is a single execution context for the basic blocks inside the loop. As the analysis bases on safe overapproximations of system states, the precision may be narrowed by creating virtual copies of nodes for different execution contexts, as the possible system states may diverge less for these nodes. Furthermore, nodes may even be split below a basic block-level to represent different execution contexts inside a basic block (e.g., due to conditional instructions).

It is important to note, that the analysis always starts at a binary- or assembly-level, where the exact machine instructions of a program are known, and not at a rather abstract level like the source code. As the aim of the analysis is to retrieve a *safe* WCET, the analysis cannot be done at a higher abstraction level, as the translation of an abstract high-level source code to the actual set of instructions to be executed is not unique, hence any possible actual execution time is unknown. While there is ongoing research in the field of so-called *early-stage WCET prediction* [BCdMS17, AGLS16], it is not further discussed as it is not an alternative to WCET analysis on a binary-level, but can rather be used for, e.g., high-level optimization steps.

In case of a joint multi-core static timing analysis, this step of creating the control-flow graph is done individually for each program.

Value Analysis. The aim of the value analysis step is to derive safe approximations of the possible system states, represented by the system's memory cells, by means of abstract interpretation. To ease the computational feasibility, the actual range of analyzed memory cells is typically restricted to the processor's registers and a small subset of the memory, such as the stack. The remaining memory cells are not analyzed and hence are assumed to contain any possible value at any given time. Results of the value analysis can have an immense influence on the precision of the timing analysis, especially in a multi-core system. Indirect memory accesses are one such example. As the address to be accessed is read from a register in the case of an indirect memory access, the precision with which the register's content during this memory access can be analyzed is crucial. If nothing can be inferred regarding the register's content, it remains unknown whether the access will be directed to a shared or a private memory, which will potentially lead to a vast overapproximation of the worst-case timing.

In case of a joint multi-core static timing analysis, this value analysis step is individually done for each program as well.

Microarchitectural Analysis. The task of the microarchitectural analysis is to derive safe upper bounds on the execution time for each node in the constructed control-flow graph, utilizing the results of the preceding value analysis. This step requires deep knowledge of a processor's architecture and the exact system (e.g., memory types, their corresponding latency or the bus arbitration policies) used. The microarchitectural analysis models all parts of a processor influencing the execution time of a program and creates a safe approximation of all possible system states during the execution. This includes the detailed modeling of available pipelines of a processor. Due to this requirement of a deep and precise knowledge of the inner workings of a processor in order to perform a safe microarchitectural analysis and therefore a safe overall analysis, static timing analyses are typically restricted to comparably simple processors. Although the means of static timing analyses can be translated to more complex architectures (e.g., Jacobs et al. [JHH16] presented an analysis framework supporting out-of-order execution pipelines), the non-existing open documentation of complex commercial processors hinders this. In these cases, a so-called *hybrid* approach can be used. A notable example for this is TimeWeaver [KPWF19] from AbsInt. Here, the program under analysis is executed on the target platform and instruction traces are captured. Using these traces, upper bounds on the execution time per basic block are derived. In contrast to a fully measurement-based method,

such a hybrid approach derives measured worst-case execution times on a basic block-level and not for the entire program. Essentially, the microarchitectural analysis is replaced by measurements here, while the other parts are performed as in a regular static timing analysis. While this does not deliver a formal guarantee for the final WCET, it increases the safety compared to a fully measurement-based approach.

In case the system under analysis features caches, the microarchitectural analysis also performs a so-called *cache analysis*. This cache analysis determines for each potential access to a cached memory of each node in the CFG whether this access will result in a cache hit, a cache miss or potentially both. The accuracy of this analysis is heavily depending on the cache's replacement policy [Rei08].

In case of a joint multi-core static timing analysis, the microarchitectural analysis is performed jointly for all cores (if required). For all bus-independent instructions (which are located in a private memory and do not perform an access to a shared memory), the analysis can derive timings without the insight of the other cores. In case an instruction is accessing a shared memory (either implicitly by being located in the shared memory or explicitly), the microarchitectural analysis analyzes if any other core might perform a shared memory access simultaneously and derives the worst-case time for this access being blocked by other cores. This joint microarchitectural analysis is performed iteratively in a loop, as the determination of which accesses may happen in parallel is done in an iterative fashion. Once converged, the analysis derived precise execution time bounds for each node in the control-flow graph, including timings influenced by interferences of other cores.

As an exception, a multi-core system featuring a TDMA-arbitrated bus can be analyzed precisely without a joint microarchitectural analysis, but with an individual analysis carried out for each core. This is due to the fact that the access behavior of one core does not influence the bus-related timings of any other core. Since in a TDMA-arbitrated bus each core has a dedicated time slot with a fixed length within a repeating schedule of slots, the number of cycles a specific access may be delayed by until it is granted is completely independent from the behavior of other cores in the system. Hence, the microarchitectural analysis can be carried out for each core individually in this case and no iterative approach is necessary.

Path Analysis. After the microarchitectural analysis, an upper bound on the execution time for each node in the control-flow graph is known. As a final step of the static timing analysis, the task of the path analysis is to find which path leads to the program's worst-case execution time (or in case of a joint WCET analysis, its worst-case response time), the so-called Worst-Case Execution Path (WCEP). A *path* refers here to a path through the control-flow graph which starts the source node (which denotes the program's entrypoint) and ends at a sink (which represents an exit of the program). One popular approach is to model the control-flow graph and its derived timing bounds per node inside an Integer Linear Program (ILP). By maximizing the number of cycles required for a path through the program, the program's WCET is found. Using this method, all possible paths through the control-flow graph are described *implicitly* by modeling the graph relationships inside an ILP, hence it is known as the Implicit Path Enumeration Technique (IPET) [LM95]. Relationships constraining the maximum allowed execution frequency of certain nodes (e.g., in nested triangular loops) can be easily expressed via linear constraints directly inside

the ILP. A potential downside to the IPET approach is the use of ILP, as the solving time increases exponentially in the worst case with the number of variables [Pap81]. Yet, especially when using powerful commercial ILP solvers, the solving times are in a reasonable time, making the IPET approach the default path analysis for many static timing analyzers like, e.g., AbsInt aiT [FH04, Ste10].

On the other hand, approaches which explicitly search the entire graph for the worst-case execution path are existing as well [KFM13, AAN11, CB02, Erm03]. In comparison to the ILP-based IPET approach, explicit graph searching algorithms typically have the advantage of a better worst-case scaling [Kle15] and can integrate detailed timing information derived from the microarchitectural analysis in a straight-forward manner (although the IPET approach can also be extended to a pipeline-state-level [Ste10]). On the downside, complex flow-restrictions (e.g., mutual exclusion of partial paths) are very hard, if not impossible, to describe in explicit path searching approaches. Additionally, as their support is mostly limited to reducible loops (having only a single entry), irreducible graphs have to be converted in the first place, increasing the complexity of the resulting graph.

Independent from the actual approach chosen to perform the analysis, the path analysis finally returns the program's worst-case execution time, or in case of a joint multi-core timing analysis, the worst-case response time.

2.2.2 Probabilistic Timing Analysis

In contrast to the previously described method of a static timing analysis, a *probabilistic* timing analysis does not aim at a provable worst-case timing estimation under all circumstances, but rather at an upper bound on the execution time within a certain confidence level. This relates to, e.g., the requirements on electronic parts in safety-critical domains, such as avionics. As electronics age and are never "perfect" due to manufacturing imprecision, it is impossible to rule out the failure of an electrical part. Therefore, certain guidelines (e.g., MIL-HDBK-217F [oD91] for avionics) are used to estimate the reliability (typically expressed as a Mean Time Between Failure (MTBF)) of electrical parts and, based on that, the overall reliability of the system. In a similar manner, probabilistic timing analysis does not guarantee a general worst-case execution time, but rather derives an upper bound on the execution time of a program depending on the required confidence level.

Overall, there are two main types of probabilistic timing analysis: Measurement-based Probabilistic Timing Analysis (MBPTA) and Static Probabilistic Timing Analysis (SPTA). In the following, both approaches are briefly introduced.

As the name suggests, MBPTA [CSH+12, HAC+15] is based on taking actual measurements of execution times of the program under analysis and uses the measurements together with statistical means to derive a so-called Probabilistic Worst-Case Execution Time (pWCET). The underlying idea of MBPTA is based on the extreme value theory (EVT) [Jen55] which is used for extreme rare occurrences, deviating greatly from the median of its statistical distribution. The concept of MBPTA enables the possibility to derive an upper bound on the execution time within a guaranteed confidence level *without* the need to model the underlying hardware architecture, as it relies on a set of measurements. The feature of not requiring a detailed model of the underlying hardware architecture can be a significant advantage over the conventional static timing analysis. As the exact inner workings and structures of nearly all

modern processors are widely unknown and not publicly documented, deriving a sound model of a processor requires a tedious amount of reverse engineering work. Additionally, even if the exact inner structure of a processor is known, modern processor features such as, e.g., out-of-order execution, speculative execution and multiple parallel pipelines, make an exact static timing analysis very complex. Emerging from the domain of research and academia, commercial tools, such as RapiTime [Rap20], are already existing, employing the concepts of MBPTA.

In contrast to MBPTA, SPTA [CQV+13, AD14] does not rely on actual measurements on the hardware. Instead, each instruction in a given program has a corresponding probability distribution of possible execution times. By combining all possible paths of a program and the corresponding sequence of instructions and hence, the individual distributions of execution time probabilities, an overall probability distribution for the execution time of the entire program can be derived. As this approach requires the knowledge of the probability distribution of each instruction, a deeper insight of the underlying hardware is necessary when compared to MBPTA. Yet, in general, less knowledge is required when compared to common static timing analyses as presented in the previous section.

On the downside, both probabilistic timing analysis approaches face potential issues concerning the requirements for a sound usage of the statistical means. Due to this, the underlying hardware is required to meet certain aspects, as otherwise the application of a probabilistic timing analysis is unsafe. One essential requirement is that the potential difference in execution time of an instruction is supposed to be random and not depending on the previous measurements or system state [HAC+15]. This requires caches to follow a *true* random replacement policy or to be turned off in general. In case of multi-core architecture, a random-based bus arbitration policy would be required. Additionally, instructions with varying execution cycles (typically multiplication or division instructions) are required to be fixed to their maximum number of cycles in order to be compliant with a probabilistic timing analysis.

2.2.3 Compositional Timing Analysis

While a joint multi-core timing analysis considers multi-core-related timing effects such as stalling due to concurrent bus accesses as a part of its microarchitectural analysis, a *compositional* timing analysis considers these "external" effects separately. The actual WCET of a task (i.e. without the interference of any other cores in the system or of other parallel tasks on the executing core) is considered as a black box. Overall, a task itself is typically only described as tuple of characteristics, such as its WCET, deadline or activation period. As a compositional timing analysis handles the system under analysis with a greater level of abstraction, it does not require a deeper knowledge on the actual single tasks. Upper bounds on multi-core timing effects are derived on a so-called *system-level* and are then added onto the corresponding task's WCET, which then is returned as its WCRT. This conceptual division between the low-level WCET analysis of a task and the derivation of its WCRT on a system-level typically enables a compositional timing analysis to handle considerably large and complex systems.

On the downside, the concept of a purely compositional timing analysis may return a more pessimistic WCRT than a joint multi-core timing analysis, or even an *unsafe* one. As a compositional timing analysis handles a task under analysis as a

complete black box, it is not aware of its actual inner structure, such as its control-flow graph. By deriving an upper bound on multi-core-related timing effects and adding this value onto the task's WCET, the resulting WCRT may be pessimistic. One possible scenario is where the WCEP of a task in isolation may not have a single bus access, yet there are some bus accesses on different paths. As a compositional timing analysis simply adds the worst-case bus stall times onto the task's WCET, the corresponding WCRT is safe, yet overly pessimistic, as it relates to an impossible execution scenario. Even worse, a resulting WCRT may be unsafe if the analyzed system is not free of timing anomalies [LS99, RWT+06]. Simply spoken, so-called timing anomalies lead to the circumstance that assuming the local worst-case for each possible instruction during analysis (e.g., a cache miss or an instruction with varying execution cycles) may not lead to the actual worst-case execution time of the program. As even comparably simple modern architectures (e.g., ARM Cortex-M4 [HJR16]) suffer from timing anomalies, the concept of compositional timing analysis is not directly applicable to many systems. In the following, the analysis concepts of two widely popular compositional timing analyses are briefly discussed.

Real-Time Calculus (RTC) is largely based on the concepts of Network Calculus (NC) by Le Boudec and Thiran [LBT01], which itself bases on the work of Cruz [Cru91]. It was first presented by Thiele et al. [TCN00] in 2000. The communication or processing demand of a task is represented using so-called event arrival functions. These event arrival functions describe, e.g., the upper bound of bus accesses of a task in the time interval domain. As a counterpart, the maximum bandwidth or processing power of a bus or processor is represented using a so-called service function, described in the time interval domain as well. By processing an event arrival function and its corresponding service function, specific characteristics, such as the maximum delay each event may experience can be derived (whereas an event can be, e.g., a bus access). Furthermore, a "remaining" service function can be calculated, representing the leftover available service after the events have been served. E.g., in case of a system where this service (e.g., bus bandwidth or processing time) is arbitrated in a fixed priority-manner, this "remaining" service function can then be used as the service function for the following task or core. By concatenating event arrival and service functions according to the system under analysis, upper bounds on, e.g., bus delays can be derived. An existing commercial timing analysis tool based on the principles of RTC is INCHRON chronVAL [INC20], whereas an open source academic tool box for MATLAB is existing as well [WT06].

The so-called SymTA/S timing analysis approach follows a slightly different concept. It was first presented by Richter et al. [Ric04, HHJ+05] in 2005. Whereas the demands of a task, such as bus accesses or execution time, are also represented using event arrival functions, the derivation of upper timing bounds and outgoing event arrival functions is performed differently. SymTA/S follows an iterative approach in which so-called local and global analyses are carried out in an alternating fashion. During a "local" analysis, each processing element (e.g., a core or a bus here) is analyzed on its own. This means that, given the corresponding event arrival functions to this processing element, maximum delays and outgoing event arrival functions are calculated. Subsequently, the system is analyzed on a "global" level where the previously derived outgoing event arrival functions are forwarded to their corresponding next element (e.g., from a bus to another core). Following this iterative

approach which stops once a fix-point solution is found, also complex systems with cyclic dependencies can be analyzed. The commercial tool SymTA/S [Sym16] is now part of Luxoft [Lux20], whereas an open source academic version pyCPA [DAE12] exists as well.

3 WCET-aware C Compiler

The WCET-aware C Compiler (WCC) is a C compiler framework with a special focus on generating and optimizing code for hard real-time systems [FL10]. It currently supports Infineon TriCore-based and ARM-based processors. In particular, WCC offers support for the Infineon TC1796 and TC1797 processors, whereas ARM7TDMI-based and Cortex-M0-based processors are supported from the ARM family, although the Cortex-M0 is currently still in progress of being fully integrated. Additionally, support for LEON3-based processors is being added currently as well. Furthermore, WCC supports compilation and optimization for a hypothetical ARM7TDMI-based multi-core system for the purpose of research [Kel15].

For many parts, WCC offers the same functionality and behavior as "regular" compilers, such as GCC [SC09] or llvm [Lat02]. A large set of standard compiler optimizations is available, such as loop unrolling, function inlining, constant folding and propagation, peephole optimization, etc. Similar to the usage of GCC or llvm, WCC offers different levels of overall optimization, which in turn automatically activate a set of optimizations. A major difference of WCC compared to "regular" compilers is the dedication to hard real-time systems and the possibility to enable optimizations which aim particularly at improving the *worst-case* timing behavior. While the previously mentioned standard compiler optimizations typically aim at reducing the average-case execution time of a program, WCC is able to perform optimization focusing on reducing the WCET. Although commonly used compilers, such as GCC or llvm, offer a plethora of possible optimizations, they are inherently not capable of optimizing for the worst-case timing behavior. This is due to the obvious fact that, in order to optimize for the worst case, the compiler needs to know *where* and *what* this worst case is. The possible execution times of a program (and hence also the actual WCET) depend not only on the basic processor architecture or its Instruction Set Architecture (ISA), but also on the *exact* hardware it is executed on. This includes the exact characteristics of memories being used in the system (e.g., latency, burst-behavior, cache configurations, etc.), interconnection properties (e.g., arbitration policy, priorities, etc.), internal characteristics and configuration of the processor (e.g., pipeline architecture, branch prediction, configuration of memory accelerators, etc.) and more. To precisely optimize code for the worst-case execution time, the compiler needs to be aware of the exact system it focuses on. While commonly used compilers offer a great variety of supported architectures, it is not possible to specify a targeted system at such a detailed level required for WCET-oriented optimizations. There are approaches to use common compilers to optimize in a WCET-oriented manner, for example by creating modified profiling information [BC18] or by trying to find the most suitable set of compiler flags in an iterative fashion [DSPD19]. Since these approaches do not change the underlying problem of the compiler being unaware of the precise details of the execution platform, the effects of these approaches vary greatly case by case. In contrast to this, WCC offers exactly this possibility to describe the targeted system on a very low level by defining its memory layout, memory

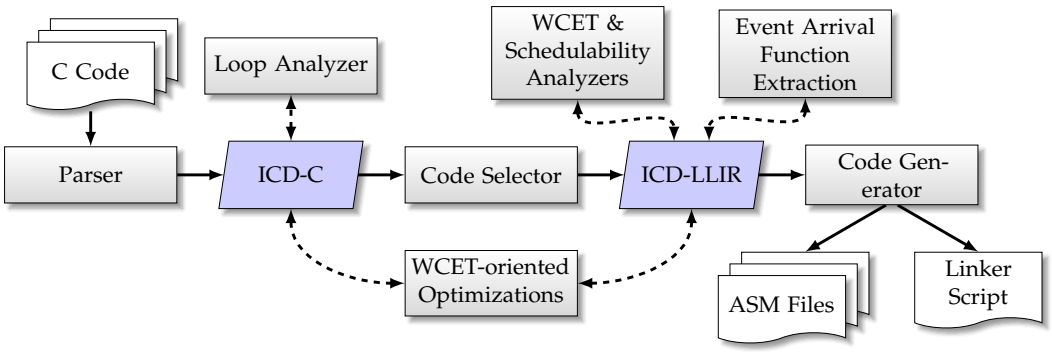


Figure 3.1. – Simplified structure of WCC, based on [FL10], updated. The solid lines indicate the flow of a regular compiler, whereas the dashed lines indicate WCC-specific parts.

properties and further required descriptions, such as bus arbitration policies and more.

On the technical side, WCC is a C cross compiler framework written in C++. At the time of writing, WCC consists of about 11 300 C/C++ files, containing around 1 762 000 lines of code. An additional 36 500 lines of code form the rules required to translate C code into hardware-specific assembly code. Since its beginnings more than 15 years ago, more than 45 authors contributed to WCC as a research and academic compiler framework.

In the following sections, an overview of WCC is given, as the concepts of this thesis were implemented as a part of this compiler framework. Therefore, a more detailed focus on the areas where parts of this thesis are located is put. A more detailed and general overview of the compiler framework is given by Falk and Lokuciejewski [FL10].

3.1 Overall Structure

In general, WCC follows the structure of a typical compiler [ALSU06, Muc98]. A simplified overview of WCC's structure is shown in Figure 3.1. The solid lines indicate the phases of a regular compiler, whereas the dashed lines indicate features which are specific to WCC. As an input, WCC receives one or multiple ANSI C files. In Figure 3.1, the lexical, syntactical and semantical analyses performed on the input files are grouped together into the parser block. The final output of the parser is the High-Level Intermediate Representation (HLIR) used inside WCC, the ICD-C [Inf09b]. The ICD-C as high-level intermediate representation is close to the original C source code and features all language concepts, such as functions, loops, variables, etc. Due to its closeness to the original source code, the ICD-C can also simply output its current content as plain C code again.

Although not explicitly depicted in Figure 3.1, a set of typical compiler optimizations can be applied to the HLIR ICD-C. This involves function inlining, loop unrolling, constant folding and propagation and several more. Beside regular compiler optimizations focusing on the average case, WCC also features WCET-directed optimizations to be performed on the high-level intermediate representation. These WCET-oriented optimizations are typically well-known compiler optimizations, such

as function inlining, yet explicitly done in a WCET-aware manner. As the actual WCET of a program is impossible to derive on such a high abstraction level, WCC offers to the possibility to "temporarily" compile the current program down to its binary, analyze its WCET and annotate low-level timing information back to the corresponding high-level constructs. This way, typical high-level optimizations can be done in a WCET-oriented manner. Additionally, WCC features a loop analyzer on the HLIR-level. In order to derive a sound WCET, an upper bound on the maximum number of iterations must be known for each loop inside the analyzed program. WCC offers a loop analyzer which is able to automatically deduce safe upper bounds on such maximum loop iterations without any further user input. While the loop analyzer is capable to derive maximum number of loop iterations in many cases, it may fail in very complex programs, where the number of loop iterations may be depending on heavy pointer arithmetic or volatile data. Therefore, WCC also offers the functionality to manually attach such loop-related information directly into the source code of the program using pragmas. These pragmas are automatically parsed and its information is attached to the ICD-C.

The code-selector transforms the rather abstract HLIR into the hardware-specific Low-Level Intermediate Representation (LLIR), the ICD-LLIR [Inf05]. Although the ICD-LLIR is a low-level intermediate representation, it is not specific to a single processor or ISA by default as it is capable to represent a wide variety of assembly languages. It offers classes to represent instructions, basic blocks, functions, etc., and methods for modification, analyses and more. An actual description of a processor's ISA then turns the ICD-LLIR into an actual processor-specific low-level intermediate presentation. The code-selector used by WCC is created using the code-generator-generator ICD-CG [Inf17]. ICD-CG is a tree pattern matcher whose set of rules to transform high-level C constructs into the equivalent assembly instructions are described using an OLIVE-compatible [Tji93] language. In case of the supported TriCore processors, this set of rules is complete and actively used for the default code selection. For the other processors supported by WCC, the rule set is being developed at the time of writing and not completed yet. Therefore, the process of code selection is being delegated to an external code-selector for these cases. The assembly code generated by the external code-selector is then being parsed in again and the corresponding low-level intermediate representation is generated.

Similar to common compilers, WCC offers a set of regular low-level optimizations which are directed to the average case, such as peephole optimization or instruction scheduling. These optimizations are applied to the ICD-LLIR. For the sake of clarity, these optimizations are not explicitly depicted in Figure 3.1. As a major difference to common compilers, WCC offers the possibility to perform WCET analyses, as well as schedulability analyses for multi-task systems. WCET analyses can be carried out using the state-of-the-art tool absInt aiT [FH04] or by using an internal analyzer. The WCET analyzer aiT is tightly coupled with WCC, enabling a fully automatic analysis run where worst-case timing information is annotated back to the ICD-LLIR (and optionally to the ICD-C). Alternatively, an internal WCET analyzer can be used. This ARM7-focused WCET analyzer was integrated in the course of the PhD thesis of Timon Kelter [Kel15]. In contrast to the WCET analyzer aiT, the internal analyzer is capable of deriving sound worst-case execution times of ARM7-based multi-core architectures. At the time of writing, a basic implementation of the internal analyzer

exists also for the TriCore 1796 inside WCC which is still under development. WCC is also able to perform timing analysis on a system-level using the Real-Time Calculus toolbox [WT06]. This coupling was integrated during the course of this thesis. Using Real-Time Calculus, comparably large and complex systems can be analyzed (cf. Section 2.2.3).

Additionally, WCC includes a schedulability analyzer for multi-task systems. This feature was added as part of the PhD thesis of Arno Luppold [Lup20]. The schedulability analyzer supports arbitrary task activation patterns and a wide range of arbitration policies. Using the outcome of the WCET or schedulability analyzers, WCC is able to perform optimizations on the low-level intermediate representation which specifically aim at improving the worst-case timing of a given system. A major part of this thesis is located at the WCET-oriented optimization on the low-level intermediate representation. One popular optimization type at this stage are memory allocation-based optimizations. Here, a memory allocation for those parts of a program is found which improve specifically its WCET.

The feature to extract so-called event arrival functions [LBT01] from the low-level representation of a program was newly added to WCC during the course of this thesis. Event arrival functions are abstract representations of specific demand requests of a program, such as bus accesses, specific function calls or a task activation itself. Using static analysis means explained further in Chapter 5, such event arrival functions can be automatically derived from a given program with an adjustable level of detail. These event arrival functions can then either be used for further timing analyses or optimizations inside WCC framework, or be exported for the use with external system analysis programs (e.g., Real-Time Calculus [WT06] or SymTA/S [Ric04]).

Finally, the code generator of WCC generates the final assembly files, as well as a corresponding linker script, which describes the location and size of used sections. Using commonly available assembler- and linker-tools, the final binary is then created.

3.2 Describing a Multi-Core System

As this thesis focuses on the optimization of multi-core systems in the hard real-time domain, the following section gives a brief overview, how a multi-core system is described for the WCC framework. The support for multi-core systems was added to WCC as a part of the PhD thesis of Timon Kelter [Kel15].

The description of a multi-core system is divided into two parts: the software- and the hardware-side. In order to compile software for a multi-core architecture, WCC supports an extended input format over a simple collection of C files, a so-called `.tasks` file. A `.tasks` file is an XML-based input file and can describe a set of C files to be compiled, their allocations to cores, as well as pointing to further configuration files. An exemplary `.tasks` file for a dual-core architecture with one task allocated to each core is shown in Figure 3.2. In this example, each task only consists of a single C file, although multiple files can be listed inside the `sources` field. The `core` tag is mandatory and sets the allocation of a task to a specific core. Optionally, each task can also be given an arbitrary name via the `name` tag, which will then be used when outputting analysis or optimization results. Using the `wccrc` tag, additional configuration files can be included, the so-called `wccrc` files. These files can contain further configurations for WCC or parts of the hardware description.

```
<task>
  <name>real_update_fixed</name>
  <core>0</core>
  <sources>
    <file>/SOME/PATH/real_update_fixed.c</file>
  </sources>
  <wccrc>/SOME/PATH/.wccrc</wccrc>
</task>
<task>
  <name>dot_product_fixed</name>
  <core>1</core>
  <sources>
    <file>/SOME/OTHERPATH/dot_product_fixed.c</file>
  </sources>
  <wccrc>/SOME/OTHERPATH/.wccrc</wccrc>
</task>
```

Figure 3.2. – An exemplary `.tasks` file for a two core architecture with one task allocated to each core.

The actual layout of the multi-core architecture is mainly described using so-called memory layout files. An exemplary excerpt of such a memory layout file for a single core is shown in Figure 3.3. This layout file defines the memory areas accessible for each core, their sizes, the access latency, the program sections to be allocated to and – essential for a multi-core system – the placement of the memory in regard to the interconnection. In the example given in Figure 3.3 two memory regions are shown, one as a part of a shared memory connected to the bus and one as a part of a private scratchpad memory. The first area named `I-SRAM-NC` is part of a slower memory which is accessible over the bus, denoted by the hierarchy `B M`, where, from the core's view, it first has to access the Bus in order to access the Memory afterwards. Each core in the system has a dedicated memory layout file which enables homogeneous, as well as heterogeneous multi-core systems.

The memory layout files themselves do not dictate any arbitration policy or configuration of the interconnection network. Such configuration can be set using a previously mentioned `wccrc` file. This way, different arbitration policies or settings, such as core priorities or an arbitration delay can be set independently from the actual memory layout using a simple configuration file.

3.3 WCET-oriented Optimizations

Since the WCC framework focuses on the automatic optimization of code for hard real-time systems, one of the most essential parts are the WCET-oriented optimizations. This section offers a brief overlook of a subset of existing WCET-oriented optimizations and the additions made during the course of this thesis.

As briefly explained in Section 3.1, WCC is able to perform WCET-oriented optimizations both on the high-level *and* low-level intermediate representations using

```
...
[I-SRAM-NC]
origin          = 0x20000000
length          = 0x80000
attributes      = RXA
cycles          = 6
buswidth        = 32
sections        = .text_uncached .text
hierarchy       = B M

[D-SP]
origin          = 0x38000000
length          = 0x40000
attributes      = RWA
cycles          = 1
buswidth        = 32
sections        = .data_spm
hierarchy       = M
...
```

Figure 3.3. – An exemplary excerpt of the memory layout of a single core.

a back-annotation which transfers worst-case timing information gathered on the low-level back to its corresponding high-level constructs. This is highly beneficial, as specific optimizations are typically better to be performed on one of the two levels. As the high-level intermediate representation inherently offers loop constructs or a more detailed level of functions and their prototypes, optimizations such as loop unrolling or function inlining are preferably performed there. Using this principle, the WCET-oriented function inlining optimization [LFS+07] available in WCC was the first one actually performing this optimization type in a WCET-aware manner. Recently, this optimization was extended [MF20] to not only focus on the program's WCET as a single objective, but rather covering multiple objectives, such as energy or average-case execution time. Another WCET-oriented high-level optimization available is loop nest splitting [FS06]. By performing this optimization, nested loop structures are split and rewritten based on the logical execution conditions of the loops, reducing the overall number of iterations and improving the analyzability of the program in terms of WCET.

A large subset of WCET-oriented optimizations available on the low-level intermediate representation are memory allocation-based ones. As it is crucial to know the exact size of the program parts to be moved to a faster memory and the potential implications on other program parts by such a move, this optimization type is performed on the low-level intermediate representation. Over the course of time, different memory allocation-based optimizations were added to WCC, each focusing on specific aspects. When aiming at the reduction of the WCET of a single-core single-task system, specific optimizations to move parts of the instructions [FK09] or data objects [FL10] to a faster, yet smaller memory can be performed. In case of a single-core multi-task system, a dedicated optimization exists to move parts of the

program code of the tasks to the faster memory specifically aiming at a schedulable system [LF17].

During the course of this thesis, three new WCET-oriented optimizations were added to WCC which are performed on the low-level intermediate representation. Two of those optimizations are memory allocation-based optimization specifically tailored towards multi-core systems. The third optimization implements the idea of traffic shaping into programs on their low-level representation to reduce the worst-case bus contention in a multi-core system. Additionally, an automatic extraction of event arrival functions on the low-level representation was added to WCC, to be used in optimizations or as an input for system-level analyses. On the technical side, these additions were added to WCC with around 10 000 lines of C++ code.

4 Low-Level Multi-Core-aware Instruction Allocation

Multi-core architectures in hard real-time domains introduced new obstacles to be overcome in terms of deriving safe upper bounds on the execution time of programs running on them. As the WCRT of a program executed on a multi-core system depends on the behavior of the shared resources and possibly also on the behavior of the concurrent cores, a timing analysis has to take these factors into account as well. While this certainly increases the complexity of timing analyses for multi-core systems, it also opens opportunities for optimizing these systems in a WCRT-oriented manner. As shared resources can easily become the bottleneck of a multi-core system and thereby heavily influence the worst-case timing behaviors, a multi-core-aware WCRT-oriented optimization can take these effects explicitly into account. Thereby, instead of simply applying a single-core-based optimization for each core in isolation and hoping that it will also work out well on the multi-core system altogether, the optimization can be tailored towards these new needs.

A powerful optimization method which has been proven to be very effective for hard real-time single-core single-task systems, as well as single-core multi-task systems, is WCRT-oriented memory allocation. This optimization approach decides which parts of a program should reside in a faster, yet smaller memory, and which should stay in a slower, yet larger one. In a multi-core system as shown in Figure 2.6 (cf. Page 13), single cores typically have a fast private memory which is very limited in size, whereas a larger shared memory exists which can be accessed over a bus. As potential blocking times while accessing the shared memory due to the bus arbitration further amplify the worst-case access latency difference between the memories, a memory allocation optimization is a promising candidate for a multi-core-aware optimization.

Beside the actual optimization technique, a crucial factor to decide is the level of abstraction used for the optimization (and inherently the timing analysis). Whereas so-called system-level analyses and optimizations model programs as tuple of a few metrics, low-level analyses on the other hand are able to handle effects even down to specific actions happening in the pipeline of a core at a specific machine instruction. While optimizations on a more abstract level typically have the advantage of a better scalability, a dedicated low-level multi-core-aware memory allocation optimization has the potential to leverage timing gains due to precise bus predictions.

In the following, a multi-core-aware static instruction allocation is presented as an example to be used to decrease the WCRT in a multi-core architecture. The optimization follows a low-level-centered approach in which it models very fine grained architectural characteristics inside the optimization problem in order to find an overall optimal solution.

The following Section 4.1 discusses related work and places this optimization into context. Specific assumptions on the multi-core architecture are then described in

Section 4.2.1. Section 4.2.2 introduces the underlying base model for instruction memory allocation. A motivational example for the potential need of a multi-core aware optimization beyond the base model is shown in Section 4.2.3. The actual extended bus-aware model is then presented in Section 4.2.4. Eventually, this bus-aware model is evaluated and compared to the bus-unaware reference approach, as well as to an Evolutionary Algorithm (EA) in Section 4.3.

4.1 Related Work

Improving characteristics of a multi-core system embedded in the hard real-time domain can be achieved by various means and on different levels of abstraction. In this section, related work on improving the worst-case response time of programs employed in multi-core systems are discussed. The focus here lies on work which in some way specifically targets multi-core architectures and takes at least some of the multi-core-related aspects into account.

Kelter [Kel15] presented two different approaches to improve the worst-case timings of a multi-core system by specifically focusing on bus-related timing effects. The first presented approach is based on the idea to tailor the bus arbitration policy and its settings specifically to the programs to be executed on the multi-core system. By employing an EA, an ideal bus arbitration policy (here namely TDMA, round robin, fixed priorities or a combination) is found, as well as the corresponding settings (e.g., slot lengths in case of TDMA or core priorities in case of fixed priorities). It is shown that by configuring the bus arbitration policy specifically towards the allocated programs, the worst-case timings can be reduced significantly. The second presented approach by Kelter is an instruction scheduling optimization for a multi-core system with a TDMA arbitrated bus. The underlying idea is to move bus-accessing instructions to better-fitting locations in terms of the TDMA schedule, such that in the best case, all bus-accessing instructions are aligned with the core's corresponding TDMA slot when being executed. Although not yielding as high improvements as the first approach, it is shown that timing improvements can be achieved by this method.

Similar to the bus arbitration policy optimization by Kelter, Rosén et al. [RAEP07] introduced an approach to improve the worst-case timings of a multi-core system by selecting a fitting TDMA policy. In this case, the authors introduced several alternative TDMA-based arbitration schemes which differ by the complexity of the repeating TDMA schedule. Their evaluation showed that by choosing a rather complex TDMA schedule which precisely reflects the demands of a given set of programs, the worst-case timings can be reduced. In a similar manner, Li et al. [LMS15] presented an approach to generate a specific TDMA schedule to reduce the worst-case response times of given programs. By analyzing the potential bus access patterns of the programs, a custom TDMA schedule is generated. Like Rosén et al., they showed that using the approach, the worst-case response time can be significantly reduced.

Liu and Zhang [LZ15] presented an ILP-based approach to reduce the worst-case timings in a multi-core system by means of scratchpad memory allocation. They proposed different SPM architectures, featuring a single private memory or a shared scratchpad memory as a second layer. Additionally, they discussed a static and a dynamic allocation technique. Though their results look promising, actual stalling times due to concurrent bus accesses do not seem to be incorporated into the evaluation or

model (or any information on the interconnection network), lowering the usefulness of the approach.

A static SPM allocation for data objects inside a TDMA bus-based multi-core system was presented by Chattopadhyay and Roychoudhury [CR11]. By using an iterative algorithm which exploits the information on potential access time windows on the variables to be allocated, an SPM allocation is found, such that the accesses to variables not placed in the private SPM are better aligned to the TDMA schedule. Any bus traffic due to instruction fetching is ignored. The evaluation showed a significant improvement using the presented algorithm.

An approach for a dynamic SPM allocation inside a multi-task multi-core system was proposed by Kafshdooz and Ejlali [KE15]. They assumed a two level SPM architecture, where each core has its own small private SPM with a very low latency. Additionally, a shared SPM exists with a slightly higher latency, which can be accessed by all cores simultaneously by individual ports. For each task, an SPM allocation is determined offline, such that during the start-phase of a task, its determined parts are loaded into the SPMs. The actual bus arbitration policy is not described and each access is simply assumed with an overall worst-case access latency. They presented an ILP-based approach, as well as one based on genetic algorithms. Overall, their ILP-based approach could significantly reduce the WCRTs of the system and yielded the best results. Yet, due to their requirement of a true n port SPM, where n is the number of cores, the approach is hard to apply to existing architectures. Similarly, Gu et al. [GZY+15] also presented an approach for task to core- and data-allocation on a multi-core multi-task system which relies on multi-port SPMs. In contrast, they assumed that each core has its own local SPM and no global SPM is existing, yet each SPM may be accessed concurrently by any of the other cores.

Goens et al. [GCOL16] presented a data allocation technique based on integer linear programming for many-core architectures. In the proposed architecture each of the processing elements has a private local SPM, whereas several shared memories are existing as well. The interconnection network is not specified in detail, but rather defined used an abstract bandwidth, whereas each task has a specific required bandwidth for each data to be allocated. It is shown that using the proposed algorithm, it is possible to find a suitable allocation in a reasonable time, whereas comparable approaches are not capable of finding a valid solution or of terminating in time.

Wasly [Was18] presented an approach for a fully SPM-oriented operating system for multi-core architectures. In this approach, each core has a local SPM which is divided into three partitions: One for the operating system and two for actual tasks being executed. Each task is always fully executed from inside the SPM. Therefore, each task has to be transferred to one of the two task partitions before being executed. Before a running task passes control to another task by either being preempted or finishing execution, a DMA transfer is initiated, to transfer the upcoming task into the other SPM partition. This scheme requires the SPM to be large enough for the two largest SPM tasks being executed, though tasks may be split up into sub-tasks if too large. Similar to this, Rouxel et al. [RSDP19] presented an approach for a combined task to core mapping and scheduling generation. Each task is executed fully from its associated local memory, whereas the transfer of the task's instructions and data is initiated beforehand. The underlying idea of this work is to arrange the schedule

of the task such that the transfer of a task should ideally not overlap with other task transfers.

An interference-aware static data memory allocation optimization for multi-core architectures was presented by Reder and Becker [RB20]. Assuming a NoC-based interconnection network with multiple shared data memories on different routers, they presented an ILP formulation to find a suitable data object allocation to reduce potential interference. The authors explored different optimization objectives and their influences on the worst-case timing, such as minimizing the overall interference costs or minimizing an approximated WCET. Significant results in terms of reduced interference costs were presented in the evaluation.

Hu et al. [HZX+14] presented a data memory allocation for a multi-core architecture with novel hybrid scratchpad memories. These hybrid scratchpad memories are closely attached to each core and consist of two different memory types: an SRAM-based memory, as well as a non-volatile memory, such as magnetic RAM or phase change memory. The idea is to combine the greater density of a non-volatile memory with the fast access of an SRAM-memory to reduce the overall access cost to a larger shared memory as much as possible. The authors proposed an algorithm to find an optimal data allocation for sub-regions of single tasks allocated to each core. As the non-volatile memory typically requires less power than an SRAM-based memory, it is shown that the optimization can also reduce energy consumption.

A method to schedule and to assign tasks onto an island-based multi-core system while also performing a code memory allocation was presented by Chang et al. [CCTF13]. Here, each so-called island consists of several cores and a local fast memory, whereas the multiple islands of the system share a global memory. The authors presented an algorithm to find an allocation of given tasks to the cores of the system, as well as a memory allocation of these tasks to achieve an overall schedulable system. Beside finding an overall schedulable system, the aim was to reduce the minimum number of required islands. The problem was then further investigated by the authors [CCC+16], exploring the complexity of the problem and presenting new approaches for different varieties of the problem. Overall, bus-related timings for accessing the shared local memory or the shared global memory were not considered here.

Kim et al. [KBCS14] presented two approaches for the SPM region mapping of functions for so-called *software-managed multi-cores*. In contrast to traditional SPM-based multi-cores, a single core inside a software-managed multi-core system can only access its private SPM and not directly the shared memory. Data or instructions have to be loaded into the private SPMs using DMA before they can be used by the corresponding cores. In case not all required parts of a program to be executed fit into the SPM, the remaining parts need to be loaded during runtime and replace other program fragments. If the replaced fragment is needed at some later point, it needs to be reloaded at that time again, increasing the program's WCRT. The work by Kim et al. discussed the problem which program parts to allocate to which parts of the SPM such that the WCRT is minimized. They presented an ILP-based approach (based on the work of Suhendra et al. [SMRC05]), as well as a heuristic. Both presented methods show significant improvements compared to ACET-oriented approaches.

Ding and Zhang [DZ11] presented different approaches for a WCRT-oriented code positioning in a dual-core system with a shared L2 cache and private L1 caches.

With one task allocated on each of the two cores, the main idea was to place the code fragments of both tasks in the memory space in order to reduce the number of cache conflicts between both cores. For this purpose, the authors developed different heuristics focusing on different strategies, such as a worst-case-oriented placement or a fairness-oriented placement. With the worst-case-oriented placement, the authors showed a WCRT reduction of up to 15%.

While a lot of related work on improving the worst-case timing behavior of a multi-core system exists, many approaches only vaguely model the timing effects of the interconnection or even completely neglect it. The few existing works which do include precise interconnection timing effects are typically based on heuristics and use timing analysis tools to derive these effects. In contrast to this, the approach presented in this chapter aims at finding an *optimal* static instruction memory allocation by precisely modeling bus-related timing effects inside an ILP. By predicting the potential side-effects of each allocation decision on a clock cycle-level, this optimization can fully take bus-related timings into account, where previous works mostly assumed pessimistic worst-case timings.

4.2 Optimization Model

With the growing complexity of functionalities to be handled by an embedded system, its required code size to realize those functions most likely increases as well. When the required code size exceeds the available private memory space of the core to which the program is allocated, at least a subset of the instructions has to be placed into the typically larger, yet shared and slower memory. Executing a program from a shared memory is especially critical, as the core needs to access the shared resource almost constantly to fetch the next instructions to be executed. If multiple cores run their code from the same shared memory simultaneously, the resulting access delays will be enormous, as the bus is operating under the highest potential load.

This scenario can be eased by either using private instruction caches or private instruction SPMs. While the use of caches is predominant in modern systems due to their transparency to the programmer, scratchpad memories are a favored choice in the domain of hard real-time systems. Caches typically feature a good average-case timing improvement, yet their influence on the estimated worst-case execution time can be far less powerful. As described in Section 2.2, a so-called cache analysis has to be carried out as a part of the static timing analysis in case caches are present in the system. The overall improvement in terms of estimated worst-case execution time therefore depends on the precision of the cache analysis, which in turn is depending on the cache properties (replacement policy, size, etc.) and the analyzed program itself.

Scratchpad memories on the other hand are not transparent to the programmer, hence memory content has to be transferred into manually, and it is addressable as a regular memory space. As the memory content is moved manually, no sophisticated analysis compared to caches has to be carried out, since it is known what content resides in the fast SPM. This removes potential timing uncertainties compared to caches. Clearly, the effectiveness of using a scratchpad memory is highly depending on *which* parts of a program are assigned to it. The allocation can be done only once during the start phase of the processor, leading to a so-called *static* allocation, or

dynamically during the runtime, resulting in a so-called *dynamic* allocation. While a dynamic allocation is potentially able to use the scratchpad memory more efficiently (as no longer used code or data can be replaced), it comes with the price of a higher overhead. In contrast to a static allocation, the loading of the SPM is done during the runtime, hence it increases the worst-case response time. The overhead of a dynamic allocation can be reduced by exploiting architectural features, such as, e.g., Direct Memory Access (DMA). Yet, to be effective, a DMA transfer has to be initiated sufficiently early before actually accessing a required program part, which needs a predictive loading. This either leads to uncertainties (hence reducing the worst-case execution time improvements) or to restrictions of the program parts to be dynamically loaded into the SPM. If the architecture does not feature DMA, the overhead of fetching and transferring the required parts of the program into the scratchpad memory have to be considered each time the allocation is changed.

In the following, a static instruction allocation is discussed. In order to maximize the benefits of an instruction allocation in a multi-core architecture, the characteristics of the shared resource have to be considered. Even worse, a multi-core-unaware allocation may even degrade the performance compared to the initial state if the accesses happen in an unfavorable manner.

This approach was first presented at the *29th Euromicro Conference on Real-Time Systems (ECRTS)* in Croatia 2017 [OLF17]. Technical implications to generate a valid memory allocation, as well as to improve the overall precision of the ILP-based model were previously presented at the *19th International Workshop on Software & Compilers for Embedded Systems (SCOPES)* in Germany 2016 [OLF16].

4.2.1 Assumptions

Before discussing the actual optimization technique, the underlying assumptions and limitations are presented.

The architecture of the system corresponds to the exemplary base architecture shown in Figure 2.6 (cf. Page 13) with N_C parallel homogeneous cores with private SPMs and shared memories which are connected to the bus. For this optimization it is assumed that the bus follows a TDMA arbitration policy, whereas the TDMA schedule consists of N_C slots. All TDMA slots are assumed to be equally-sized and to be fixed to the length of exactly one shared memory access delay. The execution of each core's task starts at a common point in time, which is assumed to be the first slot in the TDMA schedule. TDMA bus arbitration is chosen here as it enforces a temporal isolation between cores. As the bus access behavior of one core cannot influence the timing behavior of another one when using a TDMA bus arbitration, each core can be optimized individually. This is crucial for the optimization, as the goal is to precisely predict the bus-related timing effects. For work-conserving bus arbitration policies, this would require to optimize *all* cores simultaneously, as each memory allocation decision may influence any other core. The decision for all TDMA slots being equally sized and fixed to the duration of one shared memory access is done to improve the predictability. As each shared memory access of a core may only happen at a single point during the repeating bus schedule, each shared memory access serves as a kind of synchronization point when predicting the bus state.

The optimization finds a *static* memory allocation for the *instructions* of each core. As previously described, a *dynamic* memory allocation may be able to use an SPM

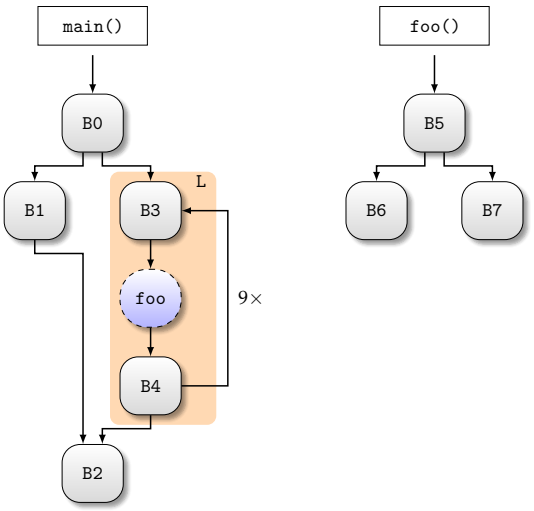


Figure 4.1. – An exemplary control-flow graph.

more efficiently, yet has the downsides of transferring overhead and being potentially less predictable. Therefore, a static memory allocation is chosen. The optimization focuses on *instructions*, as the allocation of instructions has a significantly stronger influence on the worst-case timing of a program when compared to data allocation (as each instruction, even data-accessing instructions, needs to fetched before being executed). Extending the optimization for data allocation is possible and left for future work. Nevertheless, the optimization inherently supports a given data allocation applied beforehand.

4.2.2 Bus-Unaware Base Model

The upcoming bus-aware static instruction allocation bases on the static instruction allocation presented by Falk and Kleinsorge [FK09], which in turn is based on the static data allocation presented by Suhendra et al. [SMRC05]. The optimization by Suhendra et al. is based on the ILP-based path analysis IPET first presented by Li and Malik [LM95], as it corresponds to the dual problem [BV04] of the underlying ILP formulation with added decision variables for data objects.

In the following, a brief introduction of the bus-unaware base model is given for a better understanding of the overall bus-aware optimization formulation. In general, lowercase Latin letters, such as w, are used to describe *variables* inside the ILP formulation, whereas uppercase Latin letters, such as A, denote *constants*.

An exemplary control-flow graph is depicted in Figure 4.1. Each node with a solid outline represents one basic block, whereas the rounded node with a dashed outline represents a function call. The basic idea of the model is to describe the WCEP (and hence the corresponding WCET) of the program as a minimization problem. For each basic block $\mathtt{B}_i$, a variable w_i is used to describe the worst-case execution time from basic block $\mathtt{B}_i$ to an end of the function of which $\mathtt{B}_i$ is part of. Considering the

function `foo` from the exemplary control-flow graph from Figure 4.1, the following set of inequalities is added:

$$w_{\texttt{B5}} \geq C^{+}_{\texttt{B5}} + w_{\texttt{B6}} \tag{4.1}$$
$$w_{\texttt{B5}} \geq C^{+}_{\texttt{B5}} + w_{\texttt{B7}} \tag{4.2}$$
$$w_{\texttt{B6}} \geq C^{+}_{\texttt{B6}} \tag{4.3}$$
$$w_{\texttt{B7}} \geq C^{+}_{\texttt{B7}} \tag{4.4}$$

$C^{+}_{\texttt{B5}}$ describes the worst-case execution time of a single execution of the basic block `B5`. Equation (4.1) describes the worst-case timing when starting at `B5` and taking the left path via basic block `B6`, ensuring that $w_{\texttt{B5}}$ is greater than its own WCET $C^{+}_{\texttt{B5}}$ and the accumulated WCET of the remaining path. Analogously, Equation (4.2) describes the right path via basic block `B7`. The "greater than or equal to" relational operator used in these two inequations ensures that $w_{\texttt{B5}}$ represents the longer one of both paths in terms of worst-case execution time. As basic blocks `B6` and `B7` are exiting blocks of function `foo` and therefore do not have successors in this function, their corresponding accumulated WCET w_i only consist of their net WCET C^{+}_{i}. The worst-case execution time of a single execution of function `foo` is simply described by the accumulated WCET of its entrypoint `B5`, i.e., by variable $w_{\texttt{B5}}$.

Loops are modeled using so-called meta blocks. All basic blocks belonging to a loop are forming such a meta block. This meta block's WCET is set to the worst-case execution time of a single loop iteration. Regarding the loop in function `main`, basic blocks `B3` and `B4` form the meta block `L`. Hence, the following inequalities are added to describe the WCET of this meta block and its successors, denoted as $w_{\texttt{L}}$:

$$w_{\texttt{L}} \geq 10 \cdot w_{\texttt{B3}} + w_{\texttt{B2}} \tag{4.5}$$
$$w_{\texttt{B3}} \geq C^{+}_{\texttt{B3}} + w_{\texttt{B5}} + w_{\texttt{B4}} \tag{4.6}$$
$$w_{\texttt{B4}} \geq C^{+}_{\texttt{B4}} \tag{4.7}$$

Note that the depicted loop in function `main` is tail-controlled, hence while the back edge is taken 9 times, the loop body is executed 10 times. Function calls are handled by adding the accumulated WCET variable w of the called function's entrypoint to the timing of the basic block which calls the function as can be seen in Equation (4.6). The following constraints handle the loop's meta block `L` as a regular successor of `B0`, as well as the remaining blocks of the function `main`:

$$w_{\texttt{B0}} \geq C^{+}_{\texttt{B0}} + w_{\texttt{L}} \tag{4.8}$$
$$w_{\texttt{B0}} \geq C^{+}_{\texttt{B0}} + w_{\texttt{B1}} \tag{4.9}$$
$$w_{\texttt{B1}} \geq C^{+}_{\texttt{B1}} + w_{\texttt{B2}} \tag{4.10}$$
$$w_{\texttt{B2}} \geq C^{+}_{\texttt{B2}} \tag{4.11}$$

Equation (4.8) handles the loop as a successor to `B0`, while Equations (4.9) to (4.11) are simply modeled as previously discussed. Finally, the worst-case execution time of the depicted program can be determined by minimizing the accumulated worst-case

execution time variable w of the entire program's entrypoint, i.e., of the first basic block of the function `main`:

$$\min : w_{\mathtt{B0}} \tag{4.12}$$

So far, the presented model solely performs a path analysis in order to obtain the worst-case execution time and does not do any optimization. The advantage of this specific path analysis model lies in the fact, that it is described as a minimization problem, hence it can be easily extended to decrease the WCET with subject to decision variables. In this case, a binary decision variable x_i is added for every basic block $\mathtt{B}_i$, representing whether a basic block should reside in the slow, yet large memory ($x_i = 0$), or should be moved to the fast, yet smaller SPM ($x_i = 1$). In order to estimate the benefit a basic block experiences in terms of worst-case execution time, two WCET analyses are executed before building the ILP. Before the first analysis, all basic blocks are allocated to the slow memory. This analysis returns the already discussed net WCETs per basic block C_i^+. Subsequently, all basic blocks are placed into the SPM, while the size of the SPM is virtually enlarged to fit all blocks. Virtually enlarged here means that the linker script is adapted such that a binary can be built and thereafter, a WCET analysis can be performed, yet the SPM region size exceeds the actual physical memory of the target platform. This way, a timing gain G_i per basic block $\mathtt{B}_i$ can be calculated by subtracting the basic block's WCET when being placed inside the fast SPM from the WCET when residing in the slow memory. This gain represents the ideal timing benefit per basic block when placed inside the SPM. Therefore, each partial accumulated WCET constraint is modified to include this timing gain. Equation (4.1) is used here as an example to highlight the additions:

$$w_{\mathtt{B5}} \geq C_{\mathtt{B5}}^+ - x_{\mathtt{B5}} \cdot G_{\mathtt{B5}} + k_{\mathtt{B5},\mathtt{B6}} + w_{\mathtt{B6}} \tag{4.13}$$

Beside the timing gain $G_{\mathtt{B5}}$ which is deduced from the original timing in case the basic block is allocated to the SPM, the integer variable $k_{\mathtt{B5},\mathtt{B6}}$ is added. $k_{\mathtt{B5},\mathtt{B6}}$ represents the additional number of CPU cycles (in the following simply referred to as cycles) due to additional jump correction code to be executed for the basic block transition `B5`→`B6`. A so-called jump correction can be necessary if two succeeding basic blocks (here `B5` and `B6`) are placed into different memories. This is due to the fact that, e.g., a previously relative jump instruction is not sufficient anymore due to the potentially large address offset of the two memory regions. In this case, the relative jump instruction has to be replaced by, e.g., an indirect jump instruction. This is typically more expensive timing-wise, as the destination address has to be loaded into a register beforehand. The actual repairing of jumps is done after ILP solving and memory allocation. As the additional jump correction code degrades the timing gain, it has to be considered inside the ILP. This estimation $k_{\mathtt{B5},\mathtt{B6}}$ is defined as follows:

$$k_{\mathtt{B5},\mathtt{B6}} = K_{\text{Flash}} \cdot (\overline{x}_{\mathtt{B5}} \wedge x_{\mathtt{B6}}) + K_{\text{SPM}} \cdot (x_{\mathtt{B5}} \wedge \overline{x}_{\mathtt{B6}}) \tag{4.14}$$

K_{Flash} is the estimated number of additional cycles for executing jump correction for jumping from the slow Flash memory to the SPM. Regarding Equation (4.14), this constant is added in case `B5` is placed in the shared memory ($x_{\mathtt{B5}} = 0$) and its successor `B6` is placed in the private memory ($x_{\mathtt{B6}} = 1$). This condition is described using a

logical AND operator. Analogously, K_{SPM} represents the execution time of additional jump correction code from SPM to Flash. This constant is added in case B5 is placed in the private memory ($x_{\text{B5}} = 1$) and its successor B6 is placed in the shared memory ($x_{\text{B6}} = 0$). While the instructions used for the jump correction do not depend on the memory, the required actual execution time may vary greatly, depending on from which memory the jump is executed. Therefore, the estimated jump correction costs in terms of execution time are differentiated by memory. The logical operators used in Equation (4.14) can be easily described using linear equations, see Appendix A for a detailed overview.

In order to limit the number of basic blocks allocated to the SPM, additional constraints are added to ensure the SPM is not overfilled. Here, S_{ISPM} is the total capacity in bytes of the instruction SPM, whereas S_i is the size of a basic block B_i in bytes:

$$S_{\text{ISPM}} \geq x_{\text{B0}} \cdot S_{\text{B0}} + q_{\text{B0,B1}} + q_{\text{B0,B3}} + x_{\text{B1}} \cdot S_{\text{B0}} + q_{\text{B1,B2}} + \cdots + x_{\text{B7}} \cdot S_{\text{B7}} \tag{4.15}$$

$$q_{\text{B0,B1}} = Q_{\text{B}} \cdot (x_{\text{B0}} \wedge \overline{x}_{\text{B1}}) \tag{4.16}$$

$$q_{\text{B0,B3}} = Q_{\text{B}} \cdot (x_{\text{B0}} \wedge \overline{x}_{\text{B3}}) \tag{4.17}$$

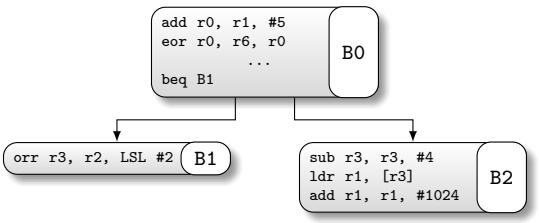
$$\ldots$$

Similar to k as the estimated number of additional cycles due to jump correction code, q denotes the additional number of bytes. E.g., if basic blocks B0 and B1 are allocated to the SPM, whereas B3 is not, additional jump correction code has to be inserted for the jump B0→B3, increasing the final amount of instructions inside the SPM. Finally, the ILP can be solved and will return an SPM allocation which minimizes the WCET of the program.

4.2.3 Motivating Example for Bus-aware Extensions

The base ILP model introduced in the previous section finds an optimal instruction SPM allocation in terms of the model for a single-core single-task system. For a multi-core single-task system, a straightforward approach would be to apply exactly the same model to each core separately. Therefore, the two timing analyses required to derive the base timings and gains per basic block are performed for each core with a timing analyzer supporting a multi-core system. As a TDMA bus arbitration is assumed, the cores are timing-wise isolated from each other, meaning an SPM allocation from one core does not influence the timing of another. In the following, a short motivating example is given, why applying the single-task single-core model may result in unexpected consequences and why considering bus effects in the ILP model can improve the found solution.

For this particular example, the exemplary control-flow graph shown in Figure 4.2(a) is assumed. Basic block B0 is not shown fully and consists of 15 instructions in total. Each instruction is assumed to have a width of 32 bit. The second instruction (`ldr`) of basic block B2 is assumed to access the `.data` section which is placed inside the shared Flash memory. As an exemplary system, we assume the architecture presented in Section 2.1.4 (cf. Page 12) with 4 parallel cores and a TDMA bus with equally-sized slot lengths (each slot can accommodate up to 5 accesses to the shared Flash memory). One Flash memory access is assumed to take 6 cycles. Furthermore, we assume that each private SPM has a size of 12 Bytes. We assume the program to



(a) Exemplary CFG II.

Basic Block	Shared	Private SPM
B0	270	15
B1	96	1
B2	114	9

(b) WCRT (in cycles) for the exemplary program.

Figure 4.2. – Exemplary control-flow graph (a) and its corresponding WCRTs per basic block (b).

be executed on core 0, which owns the first bus slot of every bus period inside the schedule.

As presented in the previous section, a static timing analysis is performed twice for the given program to generate the required gain constants G_i per basic block $\mathtt{B}_i$. In the first analysis run, all basic blocks are allocated to the shared memory, whereas during the second analysis, all basic blocks are moved to a (virtually enlarged) private SPM. The static timing analysis tool is bus-aware and includes blocking times when the core is unable to access the bus. Therefore, the timing gains derived per basic block also include possible bus-related delays. The results of these two analysis runs are shown in Figure 4.2(b). As the SPM allocation is static, the program parts are loaded into the corresponding memory parts during the boot phase of the system, hence no instructions have to be transferred to the SPM during execution time.

The ILP is set up as presented in the previous section and solved. The ILP solution will decide to place basic block `B2` into the private SPM as only basic block `B0` and `B2` are part of the worst-case execution path, yet `B0` is too large (80 B) to be allocated into the private SPM (12 B). When moving basic block `B2` into a different memory, the control-flow is not longer valid and needs to be repaired. In this case, `B2` is the fall-trough target of `B0` and is expected to be the physical successor of `B0` (in terms of memory addresses). Therefore, an additional indirect jump is included following basic block `B0` to basic block `B2`. Inside the ILP, this additional jump is considered with an additional penalty of 30 cycles. As the jump is carried out from the shared memory to the private memory, no additional code is needed in the scarce SPM. In general, this results in an expected WCRT reduction of 75 cycles, reducing the WCRT from 384 down to 309 cycles according to the ILP model. Yet, when actually applying the found allocation and performing a proper static timing analysis, the WCET is found to be 488 cycles. Thus, the expected WCRT reduction according to the ILP model heavily differs from the actual one. Even more, the "optimization" even significantly worsened the WCRT compared to the initial state. The underlying reason is found at the bus state-depending timings of specific basic blocks. In the following, the timings per basic block in relation to the bus are examined in greater detail.

Figure 4.3 depicts the worst-case execution time for the path `B0`→`B2` in case all blocks are allocated to the shared memory. In the following, a *bus slot* refers to the time slot during which a core has access to the shared bus. The overall structure, meaning the order and length of the bus slots, is referred to as the *bus schedule*. One whole TDMA cycle (consisting of N_C bus slots) is called a *bus period*, denoted by the symbol P_T in Figure 4.3. A single point in time in regard to the bus schedule is

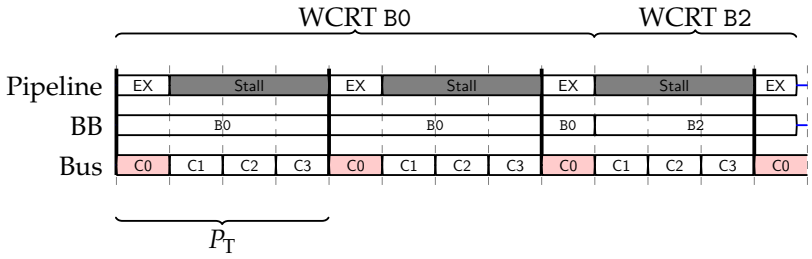


Figure 4.3. – WCRTs of basic blocks B0 and B2 in regard to the bus schedule in case all blocks are allocated to the shared memory.

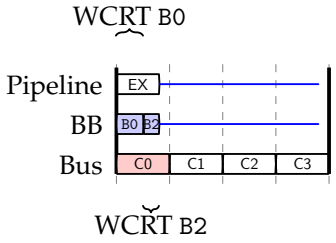


Figure 4.4. – WCRTs of basic block B0 and B2 in regard to the bus schedule in case all blocks are allocated to the private SPM.

referred to as a *bus offset*. The x-axis depicts the time, whereas the dashed vertical lines represent the single bus slots. A thick vertical line represents one whole bus period (as there are 4 cores in the exemplary system, one bus period consists of 4 slots). The lowest row labeled "Bus" shows the owner of the corresponding bus slot. The middle row "BB" shows which basic block is currently executed, whereas the top row "Pipeline" depicts the current action of the core's pipeline. As each instruction needs to be fetched from the shared memory, the pipeline stalls for the time the core cannot access the bus. Basic block B0 is executed in 3 bus slots. The succeeding basic block B2 can be executed in a single bus slot, yet the execution has to wait until core 0 is able to access the bus again.

Figure 4.4 shows the detailed execution timings in regard to the bus schedule if all basic blocks were allocated to the private SPM (with the SPM being virtually enlarged). This second step is done to derive the timing gains for each basic block. The light blue highlight in the "BB"-row depicts that the blocks are located in the SPM. It can be seen that basic blocks B0 and B2 are executed within the duration of one bus slot. Although the instructions of blocks B0 and B2 are located in the private SPM and their fetching is therefore independent from the bus schedule, the `ldr`-instruction of B2 explicitly accesses the shared memory.

Finally, Figure 4.5 shows the execution of basic blocks in regard to the bus schedule with the "optimal" SPM allocation in terms of the model found by the ILP solver. The first part of the execution of basic block B0 is identical to the execution seen in the "all in shared memory"-analysis in Figure 4.3. It then takes another bus slot to execute the additional jump code inserted to reach the SPM. As core 0 cannot access the bus immediately, it has to wait until its own slot arrives again. This drives the actual costs for the jump correction from originally assumed 30 cycles up to 120 cycles. The execution of the succeeding block B2 also differs from the execution expected due

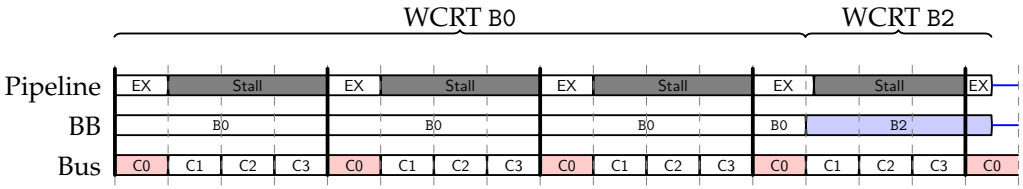


Figure 4.5. – WCRTs of basic block B0 and B2 in regard to the bus schedule with the "optimized" allocation.

to the "all in SPM"-analysis run shown in Figure 4.4. While the execution of basic block B2 immediately starts, it has to stop midway and stall. This is due to the explicit shared memory access inside the block as mentioned previously. In contrast to the "all in SPM"-analysis, basic block B2 starts its execution at the beginning of the second bus slot. While the first instruction can be fetched and executed independent from the bus state, the second instruction can be fetched, yet the actual execution has to be stalled as it explicitly accesses the shared memory. This increases the timing of B2 to 98 cycles.

Though this example featuring three basic blocks is obviously far from being a representative case for actual code used in real-time systems, and despite that the SPM capacity of 12 B does not resemble a real architecture, it showcases an underlying problem which can occur in actual systems. If, e.g., this code structure from Figure 4.2(a) is part of heavily nested loops with high execution counts, the ILP model would likely drastically over-approximate the achievable gain, while in reality potentially leading to a significantly lower timing gain or even a slightly worse timing.

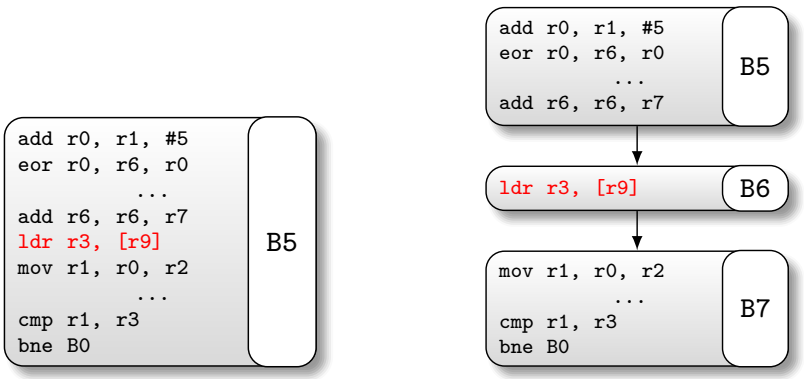
Overall, it can be seen that the misprediction of the optimization's ILP model is caused by neglecting bus-related timing effects:

- The additional jump correction code's execution time is not only depending on the bus timings, but also on the current bus state when the code is executed.
- Program fragments allocated to the private memory may still cause an access to the shared memory. As this timing is also depending on the current bus state when being executed, previous timing analysis results are unsafe to use for the calculation of timing gains if the whole program is not exactly executed as in the analysis. Since the potential bus state when trying to perform a specific shared memory access is depending on all the previously executed instructions, it is also depending on the exact memory allocation. Therefore, deriving a timing gain per basic block by simply using the timing difference between the two different allocation scenarios is insufficient, as the timing gain is potentially depending on all other allocation decisions.

In the following, an extended ILP-based model is presented which includes a prediction of bus-related timings in order to increase the optimization's accuracy.

4.2.4 Bus-aware Extensions

A simple method to somewhat predict the bus-related timings and to avoid a degradation of the WCET as just shown would be to always assume worst-case timings for all



(a) Before sub basic block splitting.

(b) With sub basic block splitting applied.

Figure 4.6. – An exemplary basic block containing an instruction with an explicit access to shared memory.

bus-related timings. Yet, this would likely decrease the accuracy of the optimization as it would be very pessimistic. As a solution, the ILP model is expanded with a set of constraints and variables to precisely predict the potential bus states when a shared memory access is executed and to derive timing bounds using these information inside the model.

The previous example showed that the timing of a bus access is depending on the current bus state when it is performed. The timing results gained from the two analysis runs therefore have to be taken into account with care:

All basic blocks allocated to the shared memory require continuous accesses to the bus, as each instruction is fetched from the shared memory. In case a program modification (e.g., additional code or basic blocks allocated to a different memory region) may cause a different bus state at the beginning of a basic block allocated to the shared memory, the worst-case timing of this basic block may differ from the "all in shared memory"-analysis. The overall timing difference compared to the previous analysis for a sequence of unmodified basic blocks due to a different bus state is bounded by the bus period P_T [Kel15]. While this already bounds the pessimism, it still introduces a potentially large penalty for every jump between the memory regions (as this would lead to a new, unknown bus state).

Similarly, the timing gained through the "all in SPM"-analysis may be uncertain for all basic blocks containing at least one explicit shared memory access in case the bus state at the start of the block's execution may differ from the one in the analysis. Here, the maximum timing difference for an unaltered sequence of basic blocks is also bounded by P_T.

Although the aim is to predict the bus state for each potential bus access as precise as possible inside the ILP model, it is clear that it will not be possible to determine the bus state at every access perfectly. This is due to, e.g., instructions with varying execution times or blocks with multiple predecessors. Therefore, the model will have to include a safe, yet pessimistic upper bound of P_T for some accesses. To decrease the magnitude of this penalty and in general the uncertainty in the predicted bus state, all bus slots are equally sized and are fixed to a minimal length, which is the access latency of the shared memory, denoted as F_S.

Furthermore, basic blocks may be split into smaller blocks beforehand to determine the bus offsets more precisely. For explicit shared memory accesses occurring in basic blocks allocated to the private SPM, the exact bus offset when performing the accesses needs to be known to predict the potential bus stall time until the access is granted. This would require timing information on an instruction-level, yet common WCET analyzers typically return timing information on a basic block-level as the smallest granularity. Basic blocks with (potential) shared memory accesses are therefore cut into so-called *sub basic blocks* before and after the memory accessing instruction. This makes sure that if a block contains an explicit memory access, it has at most one accessing instruction which is the only one in this block. Figure 4.6 shows an example for this. The basic block `B5` contains an explicit memory accessing instruction (highlighted in red). After splitting, three sub basic blocks are generated, `B5` containing all the instructions up to the memory access, `B6` consisting of only the shared memory accessing instruction, and `B7` holding all the instructions after the access. This way, precise timing information on *when exactly* the access is performed can be derived from static timing analysis tools.

As every access to the shared memory (explicit and "implicit" ones, such as fetching instructions) can only be performed at one exact bus offset (the so-called *grant cycle*) due to the TDMA schedule restriction, each access to the shared memory now serves as a kind of synchronization point. Even if the bus offset before the access is completely unknown, the bus offset after the access was performed is known cycle-accurately. In the following, sub basic blocks will be also referred to as basic blocks if not explicitly mentioned otherwise for the sake of brevity.

Despite the first impression, the rather strict assumption on the bus slot lengths does not only help on minimizing the bus state uncertainty and potential timing penalty, it also does not affect the average-case performance for the given system in a negative way. The effect of varying bus slot lengths on regular execution times was evaluated exemplarily for ARM7TDMI-based multi-core architectures (dual-, quad- and octa-core ones). Bus slot lengths were varied from the minimal length of one shared memory access latency up to 10 times the latency. Still, all cores were given equally-sized slots to represent a generic system. Overall, the results showed that, beside 4 outliers (out of more than over 4000 systems evaluated in total), none of the evaluated benchmarks and systems show a degraded ACET when using a minimum TDMA slot length (1 shared memory access per slot). The detailed results of this evaluation are presented in Appendix F.

Bus Offset Calculation The core idea to include bus-related timings in respect to an SPM allocation inside the ILP model is based on determining the possible bus states when a shared memory access is performed. By evaluating the potential bus offset (i.e., the point in time in regard to the bus schedule) of a shared memory access, an additional penalty (or also a potential timing gain) can be derived for each basic block. The former idea of analyzing the program twice, once allocated to the shared memory and once to the private memory, is kept. As basic blocks can have multiple predecessors (and thus varying bus offsets when beginning execution) and instructions with varying execution time (thus varying bus offsets when finishing execution), the bus offsets corresponding to a basic block cannot be expressed as a single scalar value. Therefore, an interval $\mathbf{o}$ is used to describe the bus offset. The

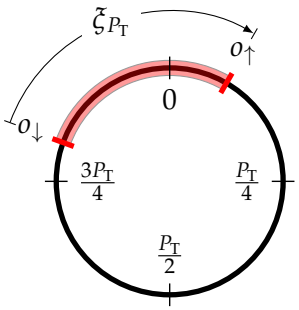


Figure 4.7. – Exemplary positive distance ξ_{P_T} between the lower offset $o_\downarrow$ and higher offset $o_\uparrow$.

interval $\mathbf{o}$ is represented by two integer variables $o_\downarrow$ and $o_\uparrow$, denoting the lower and upper bound of the interval. Both integer variables are defined in the range $[0, P_T - 1]$, denoted as the set $\mathbb{Z}_{P_T}$. Therefore, two integer variables are inserted into the ILP formulation for each offset interval $\mathbf{o}_i$ required:

$$o_{\downarrow,i}, o_{\uparrow,i} \in [0, P_T - 1] \tag{4.18}$$

The offset interval $\mathbf{o}$ is a modulo interval, hence $o_\downarrow > o_\uparrow$ may occur during a wraparound. We define the set enclosed by the modulo interval $[o_\downarrow, o_\uparrow]_{P_T}$ using the distance function $\xi_{P_T}(o_\downarrow, o_\uparrow)$, describing the distance between the offset points $o_\downarrow$ and $o_\uparrow$ as suggested by Giesen [Gie15]:

$$\xi_{P_T}(o_\downarrow, o_\uparrow) = (o_\uparrow - o_\downarrow) \bmod P_T \tag{4.19}$$

$$[o_\downarrow, o_\uparrow]_{P_T} = \{x \in \mathbb{Z}_{P_T} | \xi_{P_T}(o_\downarrow, x) \leq \xi_{P_T}(o_\downarrow, o_\uparrow)\} \tag{4.20}$$

Equation (4.19) describes the (positive) distance between the points $o_\downarrow$ and $o_\uparrow$ in the modular domain of P_T. This notion of distance between the lower offset $o_\downarrow$ and the higher offset $o_\uparrow$ with the repeating characteristic of the bus schedule represented as a ring is depicted in Figure 4.7. Using this notion of distance, Equation (4.20) defines the actual set of integers enclosed in the modulo interval $[o_\downarrow, o_\uparrow]_{P_T}$. By describing the modulo interval set using the distance function instead of splitting the interval into two regular intervals in case of a wraparound as often done [GSSS13, GNS+15], the interval can be expressed only using two integer variables instead of four inside the ILP model. The distance function as shown in Equation (4.19) can be directly described inside an ILP formulation. The required modulo-function with a constant divisor can be easily described using a set of linear constraints (cf. Appendix A for more details).

For each (sub) basic block, multiple bus offset intervals are determined inside the ILP model, $\mathbf{o}_i^{\text{In}}$ representing the incoming bus offset (meaning the possible bus offsets when the block $\texttt{B}_i$ starts execution), and $\mathbf{o}_{i,j}^{\text{Out}}$ representing the outgoing bus offset at the end of basic block $\texttt{B}_i$ to its successor $\texttt{B}_j$.

The incoming bus offset interval $\mathbf{o}_i^{\text{In}}$ of a basic block $\texttt{B}_i$ is determined depending on its memory allocation. In case the basic block resides in the shared memory, the offset interval will be set to the bus offset derived from the "all in shared memory"-analysis. As the execution of a basic block allocated to the shared memory always requires

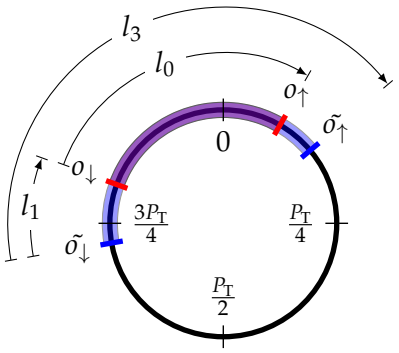


Figure 4.8. – An exemplary interval $\mathbf{o}$ and an enclosing interval $\tilde{\mathbf{o}}$ with additional distances marked.

accesses to the shared memory, the execution can only begin at the same exact cycle. This is due to the restricted TDMA schedule which allows a bus access of a core to be initiated only at one specific single cycle in the bus schedule. In case a basic block is allocated to the private SPM, the incoming bus offset interval is determined as a union of all outgoing offset intervals of the preceding basic blocks:

$$\mathbf{o}_i^{\text{In}} = \begin{cases} \mathbf{A}_{i,\text{S}}^{\text{In}} & \text{if } x_i = 0, \\ \bigsqcup\limits_{j \in \mathcal{P}_i} \mathbf{o}_{j,i}^{\text{Out}} & \text{else.} \end{cases} \tag{4.21}$$

The set $\mathcal{P}_i$ contains all basic blocks directly preceding $\mathtt{B}_i$. $\mathbf{A}_{i,\text{S}}^{\text{In}}$ is the incoming offset interval of basic block $\mathtt{B}_i$ derived from the "all in shared memory"-analysis. Equation (4.21) forces the incoming offset interval at basic block $\mathtt{B}_i$ to the analyzed interval in case the block resides in the shared memory. Note that a potential difference in the outgoing bus offset interval of a predecessor of $\mathtt{B}_i$ and $\mathtt{B}_i$'s incoming offset interval will be handled later in the form of a timing penalty separately. If $\mathtt{B}_i$ is placed into the private memory, the incoming offset interval is determined by deriving an enclosing interval of all outgoing offset intervals of all preceding basic blocks. The condition that an interval $\tilde{\mathbf{o}} = [\tilde{o_\downarrow}, \tilde{o_\uparrow}]_{P_\text{T}}$ includes another interval $\mathbf{o} = [o_\downarrow, o_\uparrow]_{P_\text{T}}$ can be defined as follows:

$$[\tilde{o_\downarrow}, \tilde{o_\uparrow}]_{P_\text{T}} \supseteq [o_\downarrow, o_\uparrow]_{P_\text{T}} \Leftrightarrow \xi_{P_\text{T}}(\tilde{o_\downarrow}, \tilde{o_\uparrow}) \geq \min\left(\xi_{P_\text{T}}(\tilde{o_\downarrow}, o_\downarrow) + \xi_{P_\text{T}}(o_\downarrow, o_\uparrow), P_\text{T} - 1\right) \tag{4.22}$$

This relationship is depicted in Figure 4.8, whereas l_0 is equal to the original interval's length $\xi_{P_\text{T}}(o_\downarrow, o_\uparrow)$ and l_1 relates to the distance of the enclosing interval's start to the enclosed interval's start $\xi_{P_\text{T}}(\tilde{o_\downarrow}, o_\downarrow)$. l_3 describes the total length of the enclosing interval, hence $\xi_{P_\text{T}}(\tilde{o_\downarrow}, \tilde{o_\uparrow})$. The proof is given in Appendix B.

The union of n offset intervals $\tilde{\mathbf{o}} = \mathbf{o}_1 \sqcup \mathbf{o}_2 \sqcup \cdots \sqcup \mathbf{o}_n$ is modeled by the following ILP constraints, ensuring that the resulting offset interval $\tilde{\mathbf{o}}$ encloses all given intervals:

$$\forall i \in \{1 \ldots n\} : \xi_{P_\text{T}}(\tilde{o_\downarrow}, \tilde{o_\uparrow}) \geq \min\left(\xi_{P_\text{T}}(\tilde{o_\downarrow}, o_{\downarrow,i}) + \xi_{P_\text{T}}(o_{\downarrow,i}, o_{\uparrow,i}), P_\text{T} - 1\right) \tag{4.23}$$

These constraints enforce that the resulting offset interval $\tilde{\mathbf{o}}$ encloses all given intervals. The second term of the min-term in Equation (4.23) serves as a maximum value, as any interval of length $P_\mathrm{T} - 1$ covers an entire bus period and hence all possible offset intervals. This is required in case the sum of interval lengths $\sum \xi_{P_\mathrm{T}}(o_{\downarrow,i}, o_{\uparrow,i})$ is greater than $P_\mathrm{T} - 1$, as otherwise the ILP would be infeasible. The min-function can easily be described using linear terms (cf. Appendix A for more details). Although the constraints added using Equation (4.23) only describe the necessary condition of an interval enclosing the values of other offset intervals, the minimization objective of the ILP will tighten the enclosing interval as much as necessary for a minimal overall WCET.

Example 4.1 (Union of Multiple Incoming Bus Offsets of a Basic Block)**.** *This example illustrates the required ILP constraints for deriving a union of incoming bus offset intervals (the second case of Equation* (4.21)*) for a distinct case. Here, the union of bus offset intervals at the beginning of the execution of basic block* `B2` *from Figure 4.1 (cf. Page 37) is exemplarily discussed.* `B2` *has two predecessors, namely* `B1` *and* `B4`*. The offset intervals at the end of the execution of the respective basic blocks just before starting to execute* `B2` *are* $\mathbf{o}^{Out}_{B1,B2}$ *and* $\mathbf{o}^{Out}_{B4,B2}$*. These two bus offset intervals are represented using two integer variables each inside the ILP formulation:*

$$o^{Out}_{\downarrow,B1,B2}, o^{Out}_{\uparrow,B1,B2} \in [0, P_\mathrm{T} - 1] \tag{4.24}$$

$$o^{Out}_{\downarrow,B4,B2}, o^{Out}_{\uparrow,B4,B2} \in [0, P_\mathrm{T} - 1] \tag{4.25}$$

The enclosing bus offset interval of both incoming offset intervals shall be named here $\tilde{\mathbf{o}}$*. It is represented using two integer variables inside the ILP as well:*

$$\tilde{o_\downarrow}, \tilde{o_\uparrow} \in [0, P_\mathrm{T} - 1] \tag{4.26}$$

The variables $\tilde{o_\downarrow}$ *and* $\tilde{o_\uparrow}$ *are then constrained using the following inequations, such that the offset interval* $\tilde{\mathbf{o}}$ *encloses both incoming offset intervals,* $\mathbf{o}^{Out}_{B1,B2}$ *and* $\mathbf{o}^{Out}_{B4,B2}$*:*

$$\underbrace{(\tilde{o_\uparrow} - \tilde{o_\downarrow}) \bmod P_\mathrm{T}}_{\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, \tilde{o_\uparrow})} \geq \min\left(\underbrace{(o^{Out}_{\downarrow,B1,B2} - \tilde{o_\downarrow}) \bmod P_\mathrm{T}}_{\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, o^{Out}_{\downarrow,B1,B2})} + \underbrace{(o^{Out}_{\uparrow,B1,B2} - o^{Out}_{\downarrow,B1,B2}) \bmod P_\mathrm{T}}_{\xi_{P_\mathrm{T}}(o^{Out}_{\downarrow,B1,B2}, o^{Out}_{\uparrow,B1,B2})}, P_\mathrm{T} - 1 \right) \tag{4.27}$$

$$\underbrace{(\tilde{o_\uparrow} - \tilde{o_\downarrow}) \bmod P_\mathrm{T}}_{\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, \tilde{o_\uparrow})} \geq \min\left(\underbrace{(o^{Out}_{\downarrow,B4,B2} - \tilde{o_\downarrow}) \bmod P_\mathrm{T}}_{\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, o^{Out}_{\downarrow,B4,B2})} + \underbrace{(o^{Out}_{\uparrow,B4,B2} - o^{Out}_{\downarrow,B4,B2}) \bmod P_\mathrm{T}}_{\xi_{P_\mathrm{T}}(o^{Out}_{\downarrow,B4,B2}, o^{Out}_{\uparrow,B4,B2})}, P_\mathrm{T} - 1 \right) \tag{4.28}$$

Equations (4.27) *and* (4.28) *correspond to Equation* (4.23)*. Equation* (4.27) *ensures that the offset interval* $\tilde{\mathbf{o}}$ *encloses the interval* $\mathbf{o}^{Out}_{B1,B2}$*, whereas Equation* (4.28) *ensures that* $\tilde{\mathbf{o}}$ *includes* $\mathbf{o}^{Out}_{B4,B2}$*. Since both inequations have to hold,* $\tilde{\mathbf{o}}$ *has to enclose* both *intervals.*

The outgoing offset interval is determined not only on a per basic block basis, but for every successor of a basic block. This is especially required to handle differing outgoing offset intervals with additional jump correction code. As a first step, the outgoing offset interval ignoring any potentially required jump correction code, named $\mathbf{o}^{\mathrm{Out}'}_i$, is derived. This outgoing offset interval is depending on the memory

allocation x_i of the block $\mathtt{B}_i$, its derived possible execution times from the analyses and whether the basic block is explicitly accessing the shared memory ($H_i = 1$). Here, not only the worst-case timing C_i^+ of a basic block $\mathtt{B}_i$ is of interest, but also its best-case execution time C_i^- (as both are required to determine the potential bus offset interval after executing basic block $\mathtt{B}_i$). These possible execution times (which can be derived from the timing analyses) are represented as the execution time interval $\mathbf{C}_i$ of basic block $\mathtt{B}_i$. An additional suffix denotes, whether the timing results from the "all in shared memory"-analysis are used ($\mathbf{C}_{i,\mathrm{S}}$) or from the "all in private memory"-analysis ($\mathbf{C}_{i,\mathrm{P}}$). The outgoing offset interval ignoring any potentially required jump correction code is set inside the ILP model according to the following equation:

$$\mathbf{o}_i^{\mathrm{Out}'} = \begin{cases} \mathbf{A}_{i,\mathrm{S}}^{\mathrm{Out}} & \text{if } x_i = 0, \\ \mathbf{A}_{i,\mathrm{P}}^{\mathrm{Out}} & \text{else if } H_i = 1, \\ (\mathbf{o}_i^{\mathrm{In}} + \mathbf{C}_{i,\mathrm{P}}) \bmod P_{\mathrm{T}} & \text{else if } \xi_{P_{\mathrm{T}}}(\mathbf{o}_i^{\mathrm{In}}) + |\mathbf{C}_{i,\mathrm{P}}| < P_{\mathrm{T}} - 1, \\ [0, P_{\mathrm{T}} - 1] & \text{else.} \end{cases} \tag{4.29}$$

In case the basic block is residing in the shared memory, the outgoing offset interval $\mathbf{o}_i^{\mathrm{Out}'}$ ignoring any potential jump correction code (named simply temporary outgoing offset interval in the following for the sake of brevity) is set to the offset interval $\mathbf{A}_{i,\mathrm{S}}^{\mathrm{Out}}$ derived from the "all in shared memory"-analysis run. In case a basic block is executed from the shared memory, the bus offset when starting its execution is always the same due to the bus slot length limitation. Therefore, also its execution time span (and thus its temporary outgoing offset interval) will always be identical. In case the basic block is allocated to the private memory, the temporary outgoing offset interval is determined differently. If the basic block contains a potential explicit access to a shared memory, the temporary outgoing offset interval is set to the offset interval derived from the "all in private memory"-analysis run $\mathbf{A}_{i,\mathrm{P}}^{\mathrm{Out}}$. As a an access to the shared memory can only be initiated during one exact cycle in the bus schedule, it serves as a synchronization point. Even if the incoming bus offset of the basic block is completely unknown, the temporary outgoing offset interval is known precisely in this case, as it will be the same as in the analysis run.

In case a basic block is allocated to the private memory and does not contain an explicit access to a shared memory, the temporary outgoing offset interval is determined using its incoming offset interval and the execution time interval derived by the "all in private memory"-analysis run $\mathbf{C}_{i,\mathrm{P}}$. The execution time interval $\mathbf{C}_i$'s lower bound is the best-case timing C_i^-, whereas its upper bound is the basic block's worst-case timing C_i^+, both derived from the timing analysis. In case the difference of the block's timings (denoted as $|\mathbf{C}_{i,\mathrm{P}}|$) plus the current length of the offset interval $\xi_{P_{\mathrm{T}}}(\mathbf{o}_i^{\mathrm{In}})$ is lower than $P_{\mathrm{T}} - 1$, the temporary outgoing offset interval is set by adding the execution time interval to the incoming offset interval and performing a modulo-operation to the base of the bus period. This condition means that the resulting offset interval will be somewhat meaningful as it will not contain the entire possible range of offsets. The addition is performed by adding the lower limits and upper limits respectively:

$$\mathbf{o}_i + \mathbf{C}_i = [(o_{\downarrow,i} + C_i^-), (o_{\uparrow,i} + C_i^+)] \tag{4.30}$$

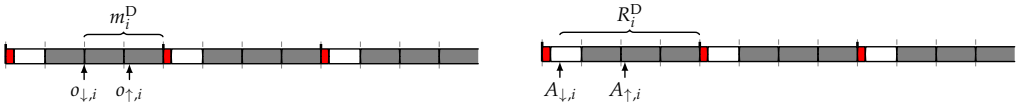
In case the basic block's worst-case and best-case timing difference plus the offset interval's current length is greater than or equal to $P_T - 1$ (the last case of Equation (4.29)), no meaningful offset prediction can be done in this case, as the possible bus offsets diverged too far. Therefore, the temporary outgoing offset interval is set to the whole possible range here.

Based on this temporary outgoing offset interval $\mathbf{o}_i^{\mathrm{Out}'}$, the actual outgoing offset interval $\mathbf{o}_{i,j}^{\mathrm{Out}}$ of basic block $\mathtt{B}_i$ when transferring control to its successor $\mathtt{B}_j$ is calculated. This offset also includes the effects of potential jump correction code. It is depending on the previously introduced temporary outgoing offset and the basic block's allocation decision:

$$\mathbf{o}_{i,j}^{\mathrm{Out}} = \begin{cases} \mathbf{o}_i^{\mathrm{Out}'} & \text{if } x_i = x_j, \\ Q_{\mathrm{S,P}} & \text{else if } x_i = \overline{x_j} = 0, \\ \mathbf{A}_{j,\mathrm{S}}^{\mathrm{In}} & \text{else.} \end{cases} \tag{4.31}$$

In case the basic block $\mathtt{B}_i$ and its successor $\mathtt{B}_j$ reside in the same memory region ($x_i = x_j$), no additional jump correction code is required, hence the outgoing offset interval is assigned the same value as the temporary offset interval. If $\mathtt{B}_i$ is assigned to the shared memory and its successor $\mathtt{B}_j$ to the private SPM ($x_i = \overline{x_j} = 0$), additional jump correction code is executed to perform this memory region-spanning jump. As the jump correction code is typically always ending with the same instruction (an indirect jump) and the TDMA slot restriction enforces each instruction executed from the shared memory as a synchronization point for the bus offset, the bus offset when performing a jump from the shared memory to the private SPM is always the same. This bus offset can be simply derived by performing a timing analysis of the jump correction code and is named $Q_{\mathrm{S,P}}$. In case the basic block $\mathtt{B}_i$ is placed in the private memory and its successor $\mathtt{B}_j$ into the shared memory (the last case of Equation (4.31)), the outgoing offset interval is known to be $\mathbf{A}_{j,\mathrm{S}}^{\mathrm{In}}$ (the incoming offset interval of $\mathtt{B}_j$ derived from the "all in shared memory"-analysis). As consequence of a jump, the processor needs to re-fill its pipeline with the first instructions of the succeeding basic block before they can actually be executed. Since these instructions are placed in the shared memory in this case, fetching them again serves as a synchronization point for the bus offset. Therefore, the outgoing bus offset interval of $\mathtt{B}_i$ is the same as the incoming bus offset interval of its successor $\mathtt{B}_j$ when being analyzed in the shared memory.

Bus-related Penalties With the help of the previously defined bus offset intervals inside the ILP model, the actual underlying motivation of predicting bus-related timing effects inside the model can be tackled. As shown in the motivational example, the bus-related timing effects can be divided into two types: Jump correction code and explicit shared memory access behavior. In case a basic block $\mathtt{B}_i$ contains an explicit shared memory access, an additional ILP variable d_i is used to describe the bus-related timing penalty (or potentially gain) dependent on the ILP's memory allocation. This variable is defined using the following equation:



(a) Bus offset interval $\mathbf{o}_i$ and the maximum stall cycles m_i^{D} as derived in the ILP model.

(b) Bus offset interval $\mathbf{A}_i$ and the maximum stall cycles R_i^{D} as extracted from the timing analysis.

Figure 4.9. – Two exemplary bus offsets $\mathbf{o}_i$ and $\mathbf{A}_i$ (where $A_{\downarrow,i}$ (resp. $A_{\uparrow,i}$) is the lower (resp. upper) bound of the analyzed offset interval $\mathbf{A}_i$) in respect to the bus schedule and their respective maximum stall cycles m_i^{D} and R_i^{D} until the next grant cycle. The processor's slot is highlighted in white, whereas the red marking depicts the grant cycle.

$$d_i = \begin{cases} m_i^{\mathrm{D}} - R_i^{\mathrm{D}} & \text{if } x_i = H_i = 1, \\ 0 & \text{else.} \end{cases} \tag{4.32}$$

m_i^{D} is an integer variable describing the maximum number of cycles required to acquire the bus grant for $\mathtt{B}_i$'s explicit shared memory access dependent on the ILP's memory allocation. R_i^{D} is the number of stall cycles accounted by the timing analysis with the "all in private memory"-configuration for the same memory access. For a better illustration of these two values, a small example is given in Figure 4.9. The two figures illustrate two offset intervals, one derived from the ILP model (Figure 4.9(a)) and one derived from the "all in private memory"-analysis (Figure 4.9(b)). It is assumed that the program under optimization is executed on the processor which owns the first slot in each bus period (highlighted). Note that this assumption is only done for this particular example and constitutes no restriction of the optimization model. The so-called grant cycle (which is the only cycle in the bus period where the processor can initiate a shared memory access), is highlighted as well. Both figures illustrate the offset interval in which a basic block $\mathtt{B}_i$ is trying to initiate a shared memory access. It is further assumed, that the ILP's current memory allocation places $\mathtt{B}_i$ into the private SPM. For Figure 4.9(a), it will take up to m_i^{D} cycles until the access is granted and can be performed. During the "all in private memory"-analysis, a different number of cycles (namely R_i^{D}) was accounted for this access to be granted in the worst case due to a different bus offset interval. As the actual memory latency for performing the memory access is identical for both cases and the expected timing gain by placing basic block $\mathtt{B}_i$ into the private SPM includes the (now seen pessimistic) timing R_i^{D}, the bus-related timing effects are described by d_i. In this case, the current ILP's memory allocation reduces the worst-case response time for $\mathtt{B}_i$'s shared memory access in comparison to the "all in private memory"-analysis.

The actual value of m_i^{D} is defined by the maximum distance between the bus offset interval when trying to initiate the shared memory access and the next grant cycle of the processor. As the basic blocks are divided into sub basic blocks, an explicit shared memory access can only be the very first (and only) instruction of a sub basic block. Therefore, the bus offset interval when trying to initiate the memory access can be derived from the incoming offset interval of the sub basic block. Using the

incoming bus offset interval $\mathbf{o}_i^{\text{In}}$, the processor's grant cycle $\gamma \in [0, P_{\text{T}} - 1]$ and an architecture-dependent constant N_A, m_i^{D} is defined using the following constraints:

$$a_{\downarrow} = \xi_{P_{\text{T}}}(o_{\downarrow,i}^{\text{In}} + N_A, \gamma) \tag{4.33}$$

$$a_{\uparrow} = \xi_{P_{\text{T}}}(o_{\uparrow,i}^{\text{In}} + N_A, \gamma) \tag{4.34}$$

$$m_i^{\text{D}} = \begin{cases} P_{\text{T}} - 1 & \text{if } \xi_{P_{\text{T}}}(o_{\downarrow,i}^{\text{In}} + N_A, \gamma + 1) \leq \xi_{P_{\text{T}}}(o_{\downarrow,i}^{\text{In}}, o_{\uparrow,i}^{\text{In}}), \\ \max(a_{\downarrow}, a_{\uparrow}) & \text{else.} \end{cases} \tag{4.35}$$

The variables $a_{\downarrow}$ and $a_{\uparrow}$ are simply used as temporary values here. $a_{\downarrow}$ describes the number of cycles between the initial shared memory access request and the next upcoming grant cycle at an offset of γ in case basic block $\texttt{B}_i$ has an incoming bus offset of $o_{\downarrow,i}^{\text{In}}$ (as $\xi_{P_{\text{T}}}(\dots)$ describes the distance between two offsets). The architecture-dependent constant N_A denotes the number of cycles between starting the execution of a memory accessing instruction and the actual memory access being performed. This constant is independent from the bus architecture and can be derived with the help of a timing analysis tool. E.g., in case of the ARM7TDMI architecture (a von Neuman architecture), the next instruction is always fetched *first* when the pipeline advances. Therefore, an explicit memory access of a load or store instruction has to wait until this instruction fetch is finished, before the memory access can be initiated. This time in between is described by the constant N_A.

The first case of Equation (4.35) covers a special situation. In case the potential offset interval when initiating the memory access spans across the offset $\gamma + 1$, the worst-case number of stalling cycles $P_{\text{T}} - 1$ is returned. A memory access request at the bus offset $\gamma + 1$ means that the only cycle in which a bus access is granted was missed by a single cycle and thus a stall of $P_{\text{T}} - 1$ cycles until the next grant cycle appears. The condition for this is simply expressed using the same way to express the set enclosed by a modulo interval as shown in Equation (4.20) (cf. Page 46). Otherwise, the second case of Equation (4.35) refines the worst-case waiting cycles by taking the maximum of the temporary values $a_{\downarrow}$ and $a_{\uparrow}$.

Beside explicit data accesses occurring during the execution of a basic block assigned to the private SPM, a second bus-related timing effect modeled using additional variables is the execution time required for jump correction code. For each successor $\texttt{B}_j$ of a basic block $\texttt{B}_i$, an ILP variable $l_{i,j}$ is introduced, describing the additional number of cycles required to execute jump correction code from $\texttt{B}_i$ to $\texttt{B}_j$ due to the current bus state. This is defined using the following equation:

$$l_{i,j} = \begin{cases} m_i^{\text{J}} & \text{if } x_i = \overline{x_j} = 1, \\ 0 & \text{else.} \end{cases} \tag{4.36}$$

m_i^{J} describes the maximum number of cycles required to acquire a bus grant after executing jump correction code at the end of basic block $\texttt{B}_i$ (which is assigned to the private SPM). Note that while Equations (4.29) and (4.31) only modeled the corresponding outgoing bus offset interval of a basic block, Equation (4.36) derives the bus-related timing costs of a jump from the private to shared memory. As a jump typically requires the pipeline of a processor to be refilled before the execution can

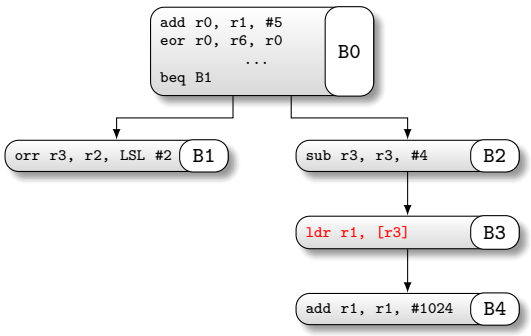


Figure 4.10. – Control-flow graph of the motivational example from Figure 4.2(a) with basic block splitting applied.

be actively resumed, a jump from the private SPM to the shared memory requires fetching instructions from the shared memory. Due to the TDMA slot restriction, each performed memory access thus serves as synchronization point after which the bus offset is precisely known. Hence, when performing a jump from the private memory to the shared memory, only the number of cycles between trying to initiate the first memory access and receiving the grant is potentially unknown. This number of cycles is described by m_i^{J}, which is defined using the following equations:

$$a'_{\downarrow} = \xi_{P_{\mathrm{T}}}(o_{\downarrow,Bi}^{\mathrm{Out}'} + K_P, \gamma) \tag{4.37}$$

$$a'_{\uparrow} = \xi_{P_{\mathrm{T}}}(o_{\uparrow,Bi}^{\mathrm{Out}'} + K_P, \gamma) \tag{4.38}$$

$$m_i^{\mathrm{J}} = \begin{cases} P_{\mathrm{T}} - 1 & \text{if } \xi_{P_{\mathrm{T}}}(o_{\downarrow,Bi}^{\mathrm{Out}'} + K_P, \gamma + 1) \leq \xi_{P_{\mathrm{T}}}(o_{\downarrow,Bi}^{\mathrm{Out}'}, o_{\uparrow,Bi}^{\mathrm{Out}'}), \\ \max(a'_{\downarrow}, a'_{\uparrow}) & \text{else.} \end{cases} \tag{4.39}$$

As can be seen, m_i^{J} is determined in a very similar manner as m_i^{D}. Equations (4.37) and (4.38) set the number of waiting cycles to acquire the bus grant after the additional jump correction code is executed and the first instruction from the shared memory needs to be fetched. Here, K_P defines the number of cycles required to execute the additional jump correction code in the private memory up to the first instruction fetch of the target basic block. Note that instead of using the outgoing bus offsets, the temporary bus offsets (meaning without any jump correction code) are used in Equations (4.37) and (4.38). Equation (4.39) follows the same structure as the previous Equation (4.35). In case the bus offset interval includes the grant cycle plus one ($\gamma + 1$), the worst-case timing $P_{\mathrm{T}} - 1$ is assumed. Otherwise, the more precise worst-case number of cycles required to receive a bus grant depending on the exact bus offset interval is used.

Final ILP Model As a short example, the final generated ILP model for the motivational example is shown in Figure 4.2(a). The adapted control-flow graph with applied basic block splitting is shown in Figure 4.10. The following constraints form the core ideas of the model, whereas the bus-sensitive parts are highlighted in red:

$$w_{\mathrm{B0}} \geq C_{\mathrm{B0}}^{+} - x_{\mathrm{B0}} \cdot G_{\mathrm{B0}} + k_{\mathrm{B0,B1}} + w_{\mathrm{B1}} + l_{\mathrm{B0,B1}} \tag{4.40}$$

$$w_{\mathtt{B0}} \geq C^{+}_{\mathtt{B0}} - x_{\mathtt{B0}} \cdot G_{\mathtt{B0}} + k_{\mathtt{B0,B2}} + w_{\mathtt{B2}} + l_{\mathtt{B0,B2}} \tag{4.41}$$

$$w_{\mathtt{B1}} \geq C^{+}_{\mathtt{B1}} - x_{\mathtt{B1}} \cdot G_{\mathtt{B1}} \tag{4.42}$$

$$w_{\mathtt{B2}} \geq C^{+}_{\mathtt{B2}} - x_{\mathtt{B2}} \cdot G_{\mathtt{B2}} + k_{\mathtt{B2,B3}} + w_{\mathtt{B3}} + l_{\mathtt{B2,B3}} \tag{4.43}$$

$$w_{\mathtt{B3}} \geq C^{+}_{\mathtt{B3}} - x_{\mathtt{B3}} \cdot G_{\mathtt{B3}} + k_{\mathtt{B3,B4}} + w_{\mathtt{B4}} + l_{\mathtt{B3,B4}} + d_{\mathtt{B3}} \tag{4.44}$$

$$w_{\mathtt{B4}} \geq C^{+}_{\mathtt{B4}} - x_{\mathtt{B4}} \cdot G_{\mathtt{B4}} \tag{4.45}$$

$$S_{\mathrm{ISPM}} \geq x_{\mathtt{B0}} \cdot S_{\mathtt{B0}} + q_{\mathtt{B0,B1}} + q_{\mathtt{B0,B2}} + x_{\mathtt{B1}} \cdot S_{\mathtt{B1}} + x_{\mathtt{B2}} \cdot S_{\mathtt{B2}} + q_{\mathtt{B2,B3}} + \cdots + x_{\mathtt{B4}} \cdot S_{\mathtt{B4}} \tag{4.46}$$

$$\min : w_{\mathtt{B0}} \tag{4.47}$$

As can be seen, the variable l is added to the timing of each basic block where additional jump correction code may be required. This variable works as an offset to the already existing variable k, which models the costs of additional jump correction costs while neglecting execution time which may vary dependent on the bus state. Additionally, a d variable is added to the timing constraint of each basic block which contains an explicit potential shared memory access. In this case, only the basic block `B3` contains an instruction accessing the shared memory.

Finally, the overall SPM size constraint and the ILP's objective function remains the same. With the additional variables and their corresponding constraints, the bus offset intervals of each basic block and each potential shared memory access are calculated dynamically inside the ILP model with respect to the chosen memory allocation. These bus offset intervals are then used to predict as precisely as possible bus-related timings dependent on which basic blocks are allocated to which memory region.

4.3 Evaluation

This section evaluates of the previously presented ILP model and discusses the achieved results. At first, the evaluation setup is introduced in Section 4.3.1. Subsequently, the actual results for varying architectures are presented in Sections 4.3.2 to 4.3.4. The optimization runtimes are evaluated in Section 4.3.5. The chapter closes with a small conclusion in Section 4.3.6.

4.3.1 Setup

The initially presented architecture shown in Figure 2.6 (cf. Page 13) is assumed as the basic architecture for the evaluation. Systems with 2, 4 and 8 cores are evaluated. The SPM of each core is resized to a certain percentage of the benchmark allocated to the core in order to see the effect of the optimization in dependence of the available memory to allocate. A TDMA arbitrated bus is assumed with each TDMA slot set to a minimal length of a single shared memory access latency. This latency is assumed to be 6 cycles (similar to the default setting of existing embedded systems [NXP09a, Inf07]), whereas an SPM access is assumed to have a latency of a single cycle. As the TDMA bus arbitration policy enforces a complete isolation between the cores and the benchmarks allocated onto them, each core can be optimized separately without any loss of precision. Therefore, each benchmark used is evaluated on its own to focus on the optimization's results, since the optimization decisions for one core do

not influence the timing of another core. As the presented optimization does not aim at allocating data objects of a program, all data objects are assumed to remain in the shared memory. This is not a strict limitation, since the data objects could be allocated to the private data SPM manually beforehand and the optimization would automatically take this into account. These data accesses would then no longer be classified as shared memory accesses. Benchmarks of the following benchmark suites were evaluated: MRTC [GBEL10], MediaBench [LPM97], StreamIt [Str18], and UTDSP [LCS92]. Additionally, a set of miscellaneous benchmarks, mostly consisting of de- and encoders were evaluated as well. In total, 81 benchmarks were evaluated for each system and configuration.

The optimization is compared against two reference approaches. The first approach is the unmodified base model presented by Falk and Kleinsorge [FK09]. This base model is shown in Section 4.2.2. An approach based on an Evolutionary Algorithm (EA) is used as a second reference. This EA-based instruction SPM allocation was presented by the author, Luppold and Falk [OLF17] in 2017 and is briefly described in Appendix C (cf. Page 255). To further inspect the effects of the optimization, the instruction allocation presented in this thesis is evaluated in two ways: Once with only the basic block splitting applied, and once with additionally the bus offset-related constraints inserted. This way, it can be differentiated by how much the timing can be decreased by only using the proposed basic block splitting and in which cases the bus offset-related constraints lead to a further improvement.

A general timeout of 1 h is used. The PISA framework [BLTZ03] is used for the EA-based approach. The maximum number of generations is set to 80 with a total of 20 individuals per generation. Mutation probability is set to 1.1% according to Greenwell [GAF95]. The influence of the mutation probability on the results of the evolutionary algorithm-based static instruction memory allocation is evaluated in Appendix G. It can be seen, that the mutation probability has hardly any influence on the quality of the results and no clear trend can be seen, which also is in line with the findings of Luppold [Lup20]. A possible reason for this is that even without any mutation, random changes are happening due to the repair function. For all mutation probabilities evaluated, the required runtime for the evolutionary algorithm is the lowest for a mutation probability of 1.0% on average. Hence, the suggested mutation probability of 1.1% seems reasonable and is used.

All experiments were performed on an Intel Xeon Server (48 cores at 3.2 GHz with 1.48 TiB RAM) and ILPs were solved using Gurobi 8.1.0, whereas each ILP solving process was limited to 4 threads. All benchmarks were compiled with several ACET-oriented optimizations activated (`-O2` optimization flag of the WCC compiler, cf. Chapter 3). Timing analyses were done using the internal WCET analyzer provided by the WCET-aware C compiler (WCC) [FL10] with its value analysis being extended by using aiT's value analyzer (version 18.10) for a more precise classification of shared memory accesses.

4.3.2 Dual-Core Evaluation

An overview of the results for the evaluated dual-core systems is shown in Figure 4.11. The y-axis of each graph represents the relative WCRT of an evaluated benchmark normalized with regard to the WCRT of the benchmark when optimized using the ILP bus-unaware base model from Section 4.2.2. Hence, a value of 1.0 corresponds to

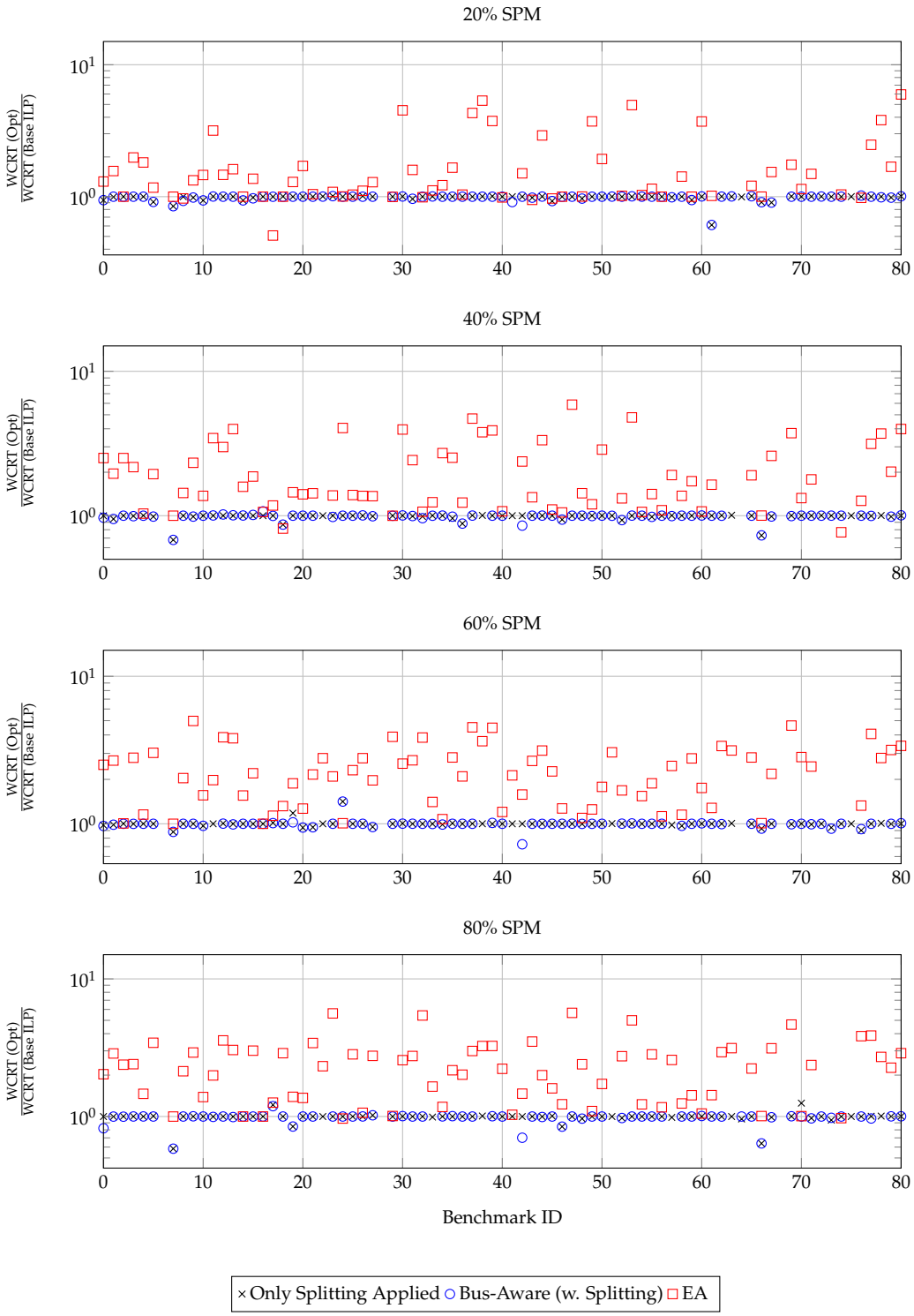


Figure 4.11. – WCRT change per benchmark when compared against the resulting WCRT of the reference ILP model for a system with 2 cores. Each graph represents an SPM configuration.

the WCRT reduction achieved by the base ILP model. A value of, e.g., 0.9 means that a WCRT reduction of an additional 10% compared to the base ILP model could be accomplished. The individual graphs represent different SPM configurations. Whereas the very top graph shows the results for the evaluation when the core's instruction SPM is resized to 20% of the benchmark's size, the second graph corresponds to the SPM resized to 40% of the benchmark's size, etc. The x-axis denotes the ID of each benchmark. For the sake of readability, the actual benchmark names are not included in this graph (the corresponding benchmark associated with each ID is listed in Appendix E.1). Figure 4.11 shows the results for three different scenarios:

1. The WCRT reduction if only the sub basic block splitting (cf. Section 4.2.4, Page 43) is applied, yet the bus-unaware base ILP model is used (labeled "Only Splitting Applied").

2. The WCRT reduction when using the fully bus-aware ILP model and sub basic block splitting applied as a requirement (labeled "Bus-Aware (w. Splitting)").

3. The WCRT reduction for the evolutionary algorithm (labeled "EA").

Since the sub basic block splitting refines the granularity of the memory blocks to be allocated, the effects of this refinement and the bus-unaware base ILP model is evaluated as well. This graph gives an overview of the general evaluation results, a more detailed insight into these results is given subsequently. In case an optimization was canceled due to a timeout, this is represented in the graph by the absence of the corresponding marker.

Overall, it can be seen that the approach based on EA yields virtually always worse results when compared to the ILP-based optimizations, independent from the SPM configuration. Only in a very few cases among all benchmarks and SPM configurations, the EA was able to achieve a better WCRT than the ILP-based optimizations. The EA is potentially able to yield better results compared to the ILP-based optimizations, as these are only optimal in regard to their model. As all ILP models used in this evaluation are not able to reflect the WCRT with the precision of a proper complete WCRT analysis, the EA-based approach can potentially achieve better results as it employs fully-fledged WCRT analysis on the actual potential solution for each individual. Compared to this, the ILP model can only estimate the additional costs of jump correction code, as well as it does not include fine-grained timing aspects as the timing analysis does (e.g., timing contexts inside a basic block due to conditional execution of instructions). On average among all SPM configurations, the EA-based approach yields a WCRT of a factor 2.1 worse compared to the WCRT of the base ILP model. In a more detailed look, this results from an average factor of 1.7 for a 20% SPM configuration, 2.1 for 40%, and 2.3 for both 60% and 80%. These comparably poor results for the EA are in line with the conclusions of Luppold [Lup20], who evaluated this approach for multi-task single-core systems. Despite the time-consuming worst-case timing analysis required for the fitness evaluation of each individual, the EA advances over many generations on average before terminating due to the timeout. Overall, the EA is able to evaluate 67.2 of maximum 80 generations on average. Despite advancing over many generations, the evolutionary approach does not converge to the same quality of solutions as the ILP model does. A possible reason for this is that the quality of potential solutions is very susceptible to even very small

changes. Changing a single basic block's allocation of a previously excellent solution may easily deteriorate the solution's fitness. Due to this circumstance, the EA can find a *good* instruction allocation very fast (typically in the first 10 generations), but can hardly advance beyond this.

It can be seen that for the most benchmarks and SPM configurations being evaluated, the bus-aware ILP-based approach presented in thesis comes to the same result as the base ILP model. On average among all SPM configurations and benchmarks, the fully bus-aware ILP model with basic block splitting applied as a prerequisite results in a 1.6% lower WCRT than the base ILP model. On a closer look, this results from an average WCRT reduction of 1.8% for the 20% SPM configuration, 1.7% for 40%, 0.6% for 60% and 2.1% at 80%. These comparably low average improvements in the WCRT when compared to the base ILP model lead to the conclusion, that for the most benchmarks and SPM configurations, the optimal program allocation is independent from bus-related timing effects (or are simply in line with them). Furthermore, the large net memory access latency difference between the shared memory and private SPM (6:1) is further amplified with the TDMA-arbitrated bus. Therefore, the timing gain of placing basic blocks inside the private SPMs easily overshadow possible under-estimations of, e.g., jump correction costs or explicit data accesses. Yet, it can also be seen that in specific cases, the bus-aware ILP can achieve significant improvements when compared to the base ILP. The dip of the average WCRT reduction at a relative SPM size of 60% seems to be inherited from the bus-unaware base optimization and will be discussed later, as it is present in all evaluated systems. Among all SPM configurations for the dual-core system, the bus-aware ILP model can achieve a further WCRT reduction of up to 41.6%.

Figure 4.12 shows a more detailed look at a subset of the evaluation results. The improvements in WCRT compared to the base ILP model are likely caused by the combination of the two proposed additions or modifications: The introduced approach of basic block splitting (as a prerequisite to a better bus offset calculation inside the ILP) already increases the number of potential allocations, as the basic blocks can be allocated in a smaller granularity. Additionally, the basic block splitting already implicitly improves the timing predictions of the bus-unaware base model, as basic blocks are split at right before a data access, turning them into "synchronization points" for the bus offsets. Therefore, even without modeling the bus offsets, the base ILP model including sub basic block splitting is expected to estimate the timing better. In addition, the bus-timing-related constraints and variables should further improve the precision of the WCRT prediction inside the ILP model, as they derive bus-related timing effects. This closer look shows the influence of both additions. Figure 4.12 shows the WCRT improvements for a subset of benchmarks when only applying the proposed basic block splitting, but not adding the bus-relating additions and once with the complete bus-aware model. Only those benchmarks whose WCRT differ by more than 1% between the two approaches are shown in the diagram. The 1% is simply chosen as a significance threshold.

Similar to the previous set of graphs, Figure 4.12 shows the results for different SPM configurations. The y- and x-axis are the same as before, yet with the benchmark names now being actually labeled on the x-axis due to more space. Overall, it is noticeable that for most cases, the ILP with the added bus-related constraints yields program allocations leading to the same (or very similar) WCRT as with only the basic

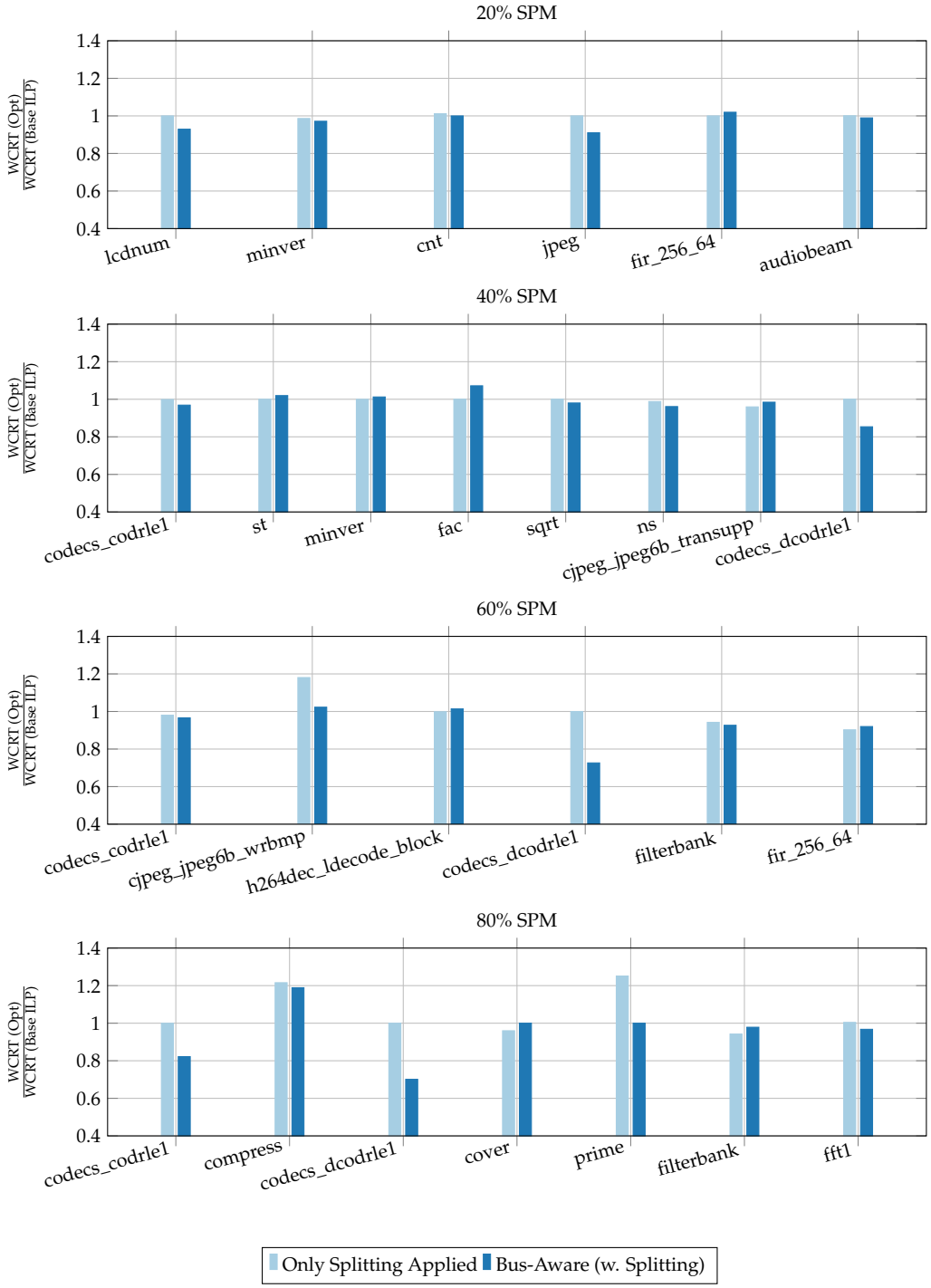


Figure 4.12. – WCRT change per benchmark when compared against the resulting WCRT of the reference ILP model for a system with 2 cores. A closer look at benchmarks whose WCRT improvement by only applying basic block splitting vs. further adding bus-timing constraints differ by more than 1% (ILP-based approaches only).

block splitting applied. This observation goes in line with the previous conclusion, that in most cases the optimal basic block allocation is independent from the exact bus-timings, as the overall timing differences overshadow smaller false estimations. Yet, for specific benchmarks and SPM configurations, the bus-timing-related constraints and variables may achieve a significant WCRT improvement over the basic block splitting approach let alone. Using a 20% SPM configuration, for two benchmarks a further WCET reduction of more than 7% can be achieved by using the bus-related timing additions. By increasing the available SPM space, the effectiveness of the additional bus-related constraints increases the magnitude of WCRT improvement. The WCRT compared against only using basic block splitting improves by up to 14.6% for a 40% SPM configuration, up to 27.4% for 60% SPM and up to 30% for 80% SPM. This increasing (maximum) improvement with a growing SPM size could hint to an increasing potential advantage of the bus-related constraints with a greater SPM size. A possible explanation for this could be that for a small SPM configuration, the potential basic blocks to be allocated there to achieve the greatest timing reduction are very few (e.g., blocks in deeply nested loops). As the WCRT reduction achieved by these few basic blocks is relatively big, the possible harm by a not ideal timing estimation is more likely to be negligible. With a larger SPM size, the possible number of basic blocks to allocate to effectively reduce the timing increases and with it the chances, that a bus-related timing could make a difference. Overall, it seems that especially data-focused benchmarks can take advantage of the bus-aware constraints, as the benchmarks with the largest improvements are all de- or encoding applications.

In a few cases the bus-related timing additions lead to a slightly worse WCRT when compared to the basic block splitting only. The largest negative deviation among all SPM configurations evaluated for the dual-core system is 7.2% at a 40% SPM setting. This is most likely due to approximations inside the bus offset derivation of the ILP model. While the ILP models the potential bus offsets as a single modulo interval, the WCRT timing analysis can represent multiple bus offset intervals.

Example 4.2. *In case a basic block has two predecessors, one with a bus offset of exactly* 0 *and the other with a bus offset of exactly* 23*, the multi-core timing analysis will represent the incoming bus offset as a set of two intervals, namely* $\{[0,0],[23,23]\}$*. As the bus offset derivation inside the ILP works on a single modulo interval per offset, the resulting interval offset inside the ILP would be* $[0,23]$*, potentially leading to a greater worst-case timing.*

Another potential pessimism is introduced by execution contexts of basic blocks modeled by the timing analysis, e.g., due to conditional instructions. These effects may cause the bus-aware ILP model to be pessimistic and to estimate a higher cost for certain allocation decisions than actually resulting. Therefore, the bus-aware additions may find a less good solution in some cases when compared to the basic block splitting only approach.

In some rare cases, the base ILP model fails to properly describe the worst-case execution path. This occurs at the `compress` benchmark. Since the base model is already flawed here, this problem is also inherited by the bus-aware ILP model. This actually leads to worse optimization results when basic block splitting is applied and also with the full bus-aware ILP model. Due to the basic block splitting, *more* blocks are assigned to the SPM, which are actually not decreasing the worst-case response time, but are increasing it due to the jump correction code. This problem is reserved for future work.

4.3.3 Quad-Core Evaluation

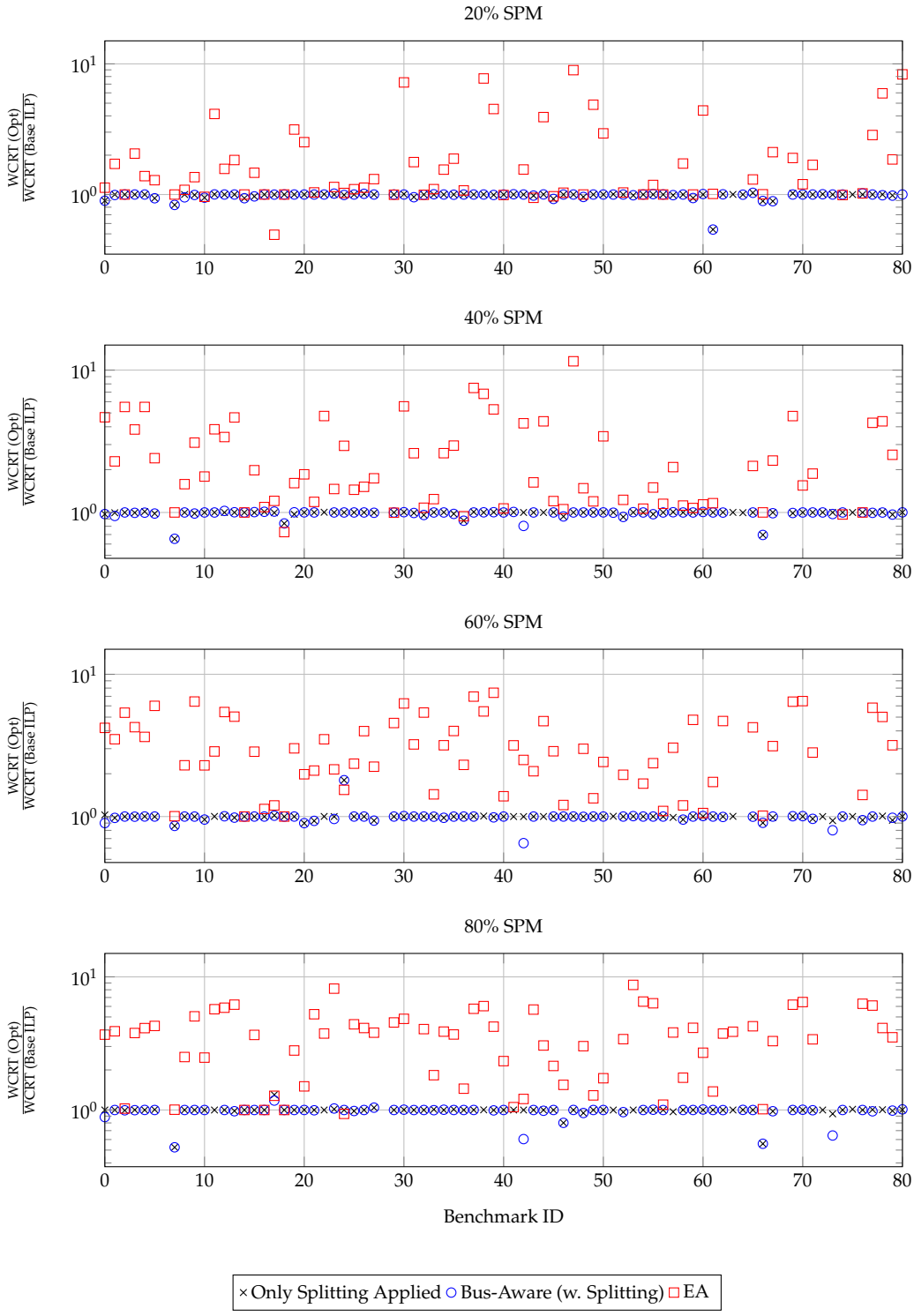


Figure 4.13. – WCRT change per benchmark when compared against the resulting WCRT of the reference ILP model for a system with 4 cores. Each graph represents an SPM configuration.

Figure 4.13 shows an overview of the evaluation results for a quad-core system with different SPM size configurations. The structure of the graphs are the same as in the previous section with the benchmark IDs on the x-axis and the normalized WCRT per benchmark and approach on the y-axis. Similar as before, the EA reference approach yields worse results than the base ILP for most of the evaluated settings. On average among all SPM configurations, the evolutionary approach results in a WCRT worse by a factor of 2.8 when compared to the base ILP. When having a closer look at the average results of the evolutionary approach depending on the SPM size, a trend can be noticed. With an average WCRT increase of a factor of 2.0 for an SPM configuration of 20%, 2.6 for 40%, 3.2 for 60% and 3.6 for 80%, the evolutionary approach tends to produce worse results with an increasing SPM size. This can be linked to the overall greater pool of possible solutions. In case of a small SPM setting of 20%, only a small fraction of the program can fit inside the SPM and hence the number of valid solutions is far lower. Additionally, the evolutionary approach yields on average worse results for the quad-core system than compared to the dual-core system. This can be explained by the fact that a single wrong allocation decision has a stronger impact in the quad-core system, as the worst-case bus stalling time is twice as long. Yet, similar as seen in the results for the dual-core system, the evolutionary approach is able to create the best allocations for a very few cases.

The bus-aware ILP approach yields similar results for the quad-core systems as seen previously for the dual-core ones. In most cases, the bus-aware ILP will return the same allocation as the bus-unaware base model does. On average over all SPM configurations and benchmarks, the bus-aware ILP model can achieve a WCRT reduction of 2.0%, which is a slight increase compared to the evaluated dual-core systems. Corresponding to the actual SPM configurations, the bus-aware ILP model achieves on average a WCRT reduction of 1.9% for a 20% SPM setting, 2.0% for 40% SPM, 0.9% for 60% SPM and 2.9% for an 80% SPM setting. These comparably low average WCRT improvements lead to the similar conclusion as seen for the dual-core systems, that for the most settings and benchmarks, the optimal allocation can be found irrespective of the bus timings. Yet, for some specific combinations also in a quad-core system, the bus-aware ILP model can improve the timing significantly.

Figure 4.14 shows a closer look at the evaluation results for a subset of benchmarks, highlighting the effects of the bus-related additions. Similar as for the evaluated dual-core systems, the benchmarks depicted in this more detailed overview differ in their WCRT using the complete bus-aware ILP model against the basic block splitting alone by more than 1%. The overall picture is very similar to the evaluated dual-core system. For most cases, the complete bus-aware model will return the same (or a very similar) allocation than when only using the basic block splitting. Yet for some specific configurations, taking into account the bus-related timings can result in a significant further WCRT improvement. For an SPM setting of 20%, the bus-aware model can achieve a WCRT improvement of up to 4.9% when compared to basic block splitting alone, 19.6% for a 40% SPM setting, 34.9% for a 60% SPM setting and up to 39.5% for an 80% SPM setting. As already seen in the evaluation of the dual-core systems, the potential influence of bus-related timing effects onto the best program allocation seems to increase with a larger SPM setting. Similar to the results of the dual-core systems, the benchmarks with the largest improvements due to the bus-aware constraints are all data-focused. A special case can be seen for the 20%

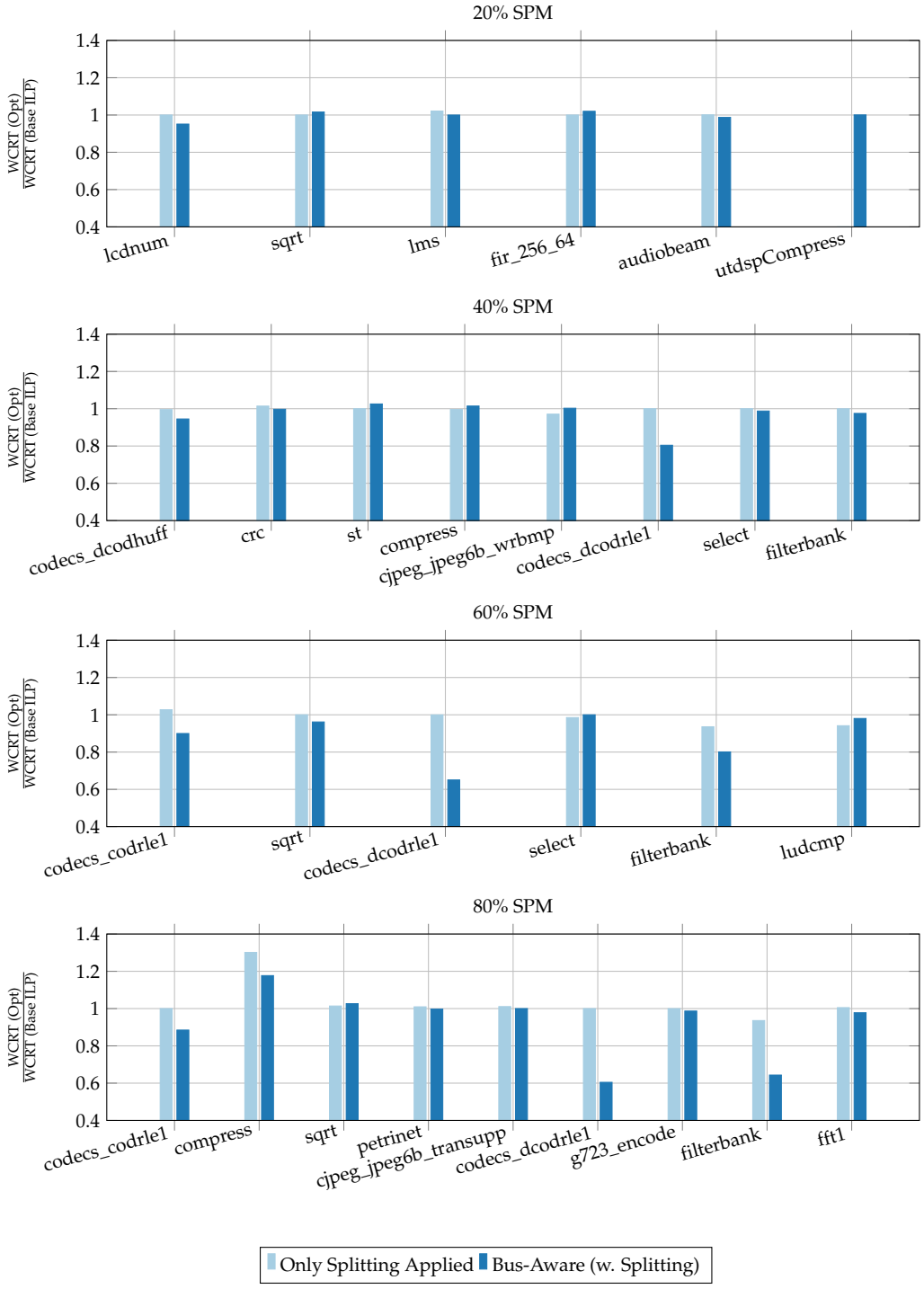


Figure 4.14. – WCRT change per benchmark when compared against the resulting WCRT of the reference ILP model for a system with 4 cores. A closer look at benchmarks whose WCRT improvement by only applying basic block splitting vs. further adding bus-timing constraints differ by more than 1% (ILP-based approaches only).

SPM setting at the *utdspCompress* benchmark. Only the result for the ILP model with bus-aware extensions is shown, as the base ILP model with basic block splitting leads to a memory allocation which caused the timing analyzer to be stuck in an infinite loop. This anomaly is subject for future investigations inside the internal timing analyzer of the WCC. Overall, the additional timing improvements by the bus-aware ILP model are on average higher for the quad-core systems than compared to the dual-core ones. This can be explained by the stronger influence of the bus-related timings, as a single unfavorable allocation may have a stronger influence due to longer bus stall times.

As already seen in the dual-core evaluation, for a few benchmark and SPM configuration combinations, the bus-aware constraints lead to a slightly larger WCRT compared to the base model with basic block splitting. Again, this can be explained by some pessimism of the bus-aware ILP model, leading to a worse timing prediction than actually resulting from a certain allocation.

4.3.4 Octa-Core Evaluation

Figure 4.15 shows an overview of the evaluation results for systems with 8 cores with varying SPM sizes for the different approaches. The structure of the graphs is identical to the corresponding graphs of the previous dual- and quad-core evaluations. Similar as in the previous evaluations, also for the octa-core systems, the evolutionary approach yields slightly better results compared to the base ILP model in a very few cases, yet worse for the most cases. On average over all SPM configurations and benchmarks, the evolutionary approach returns an allocation whose WCRT is worse by a factor of 3.4. When having a closer look at the SPM configurations, the evolutionary approach yields a larger WCRT on average by a factor of 2 for a 20% SPM, by a factor of 3 for a 40% SPM, 3.9 for a 60% SPM and 4.5 for an 80% SPM. These results go in line with the insights gained from the dual- and quad-core evaluations: The larger the SPM size gets, the more the results from the evolutionary approach deteriorate as the pool of possible solutions grows. Similarly, the quality of the SPM allocations found by the evolutionary approach also decreases with a rising number of cores, as the potential significance of a single bad allocation decision increases due to the higher bus stalling time.

Likewise, the results for the bus-aware ILP model are very similar to those for the evaluated quad-core systems. For the majority of the benchmarks, the program allocation found by the bus-aware ILP is identical to the one found by the base ILP model. On average over all benchmarks and SPM configurations, the bus-aware ILP model returns an allocation whose WCRT is 1.9% lower than compared to the base ILP. In case of a 20% SPM setting, the bus-aware ILP model improves the WCRT on average by 1.8%, in case of a 40% SPM setting by 2.5%, for a 60% SPM setting by 0.4% and in case of an 80% SPM setting by 2.8%. These improvements are very similar to the ones obtained during the quad-core evaluations. The consistent drop of average WCRT reduction observed for dual-, quad- and octa-core systems at a relative SPM size of 60% seems to be inherited from the bus-unaware base ILP model. To investigate this, the bus-unaware base ILP model was evaluated for varying SPM sizes, once with sub basic block splitting applied and once without. The detailed results are presented in Appendix H (cf. Page 275). Here, single-core single-task systems were used to rule out the influence of the bus. The results show that on average, the

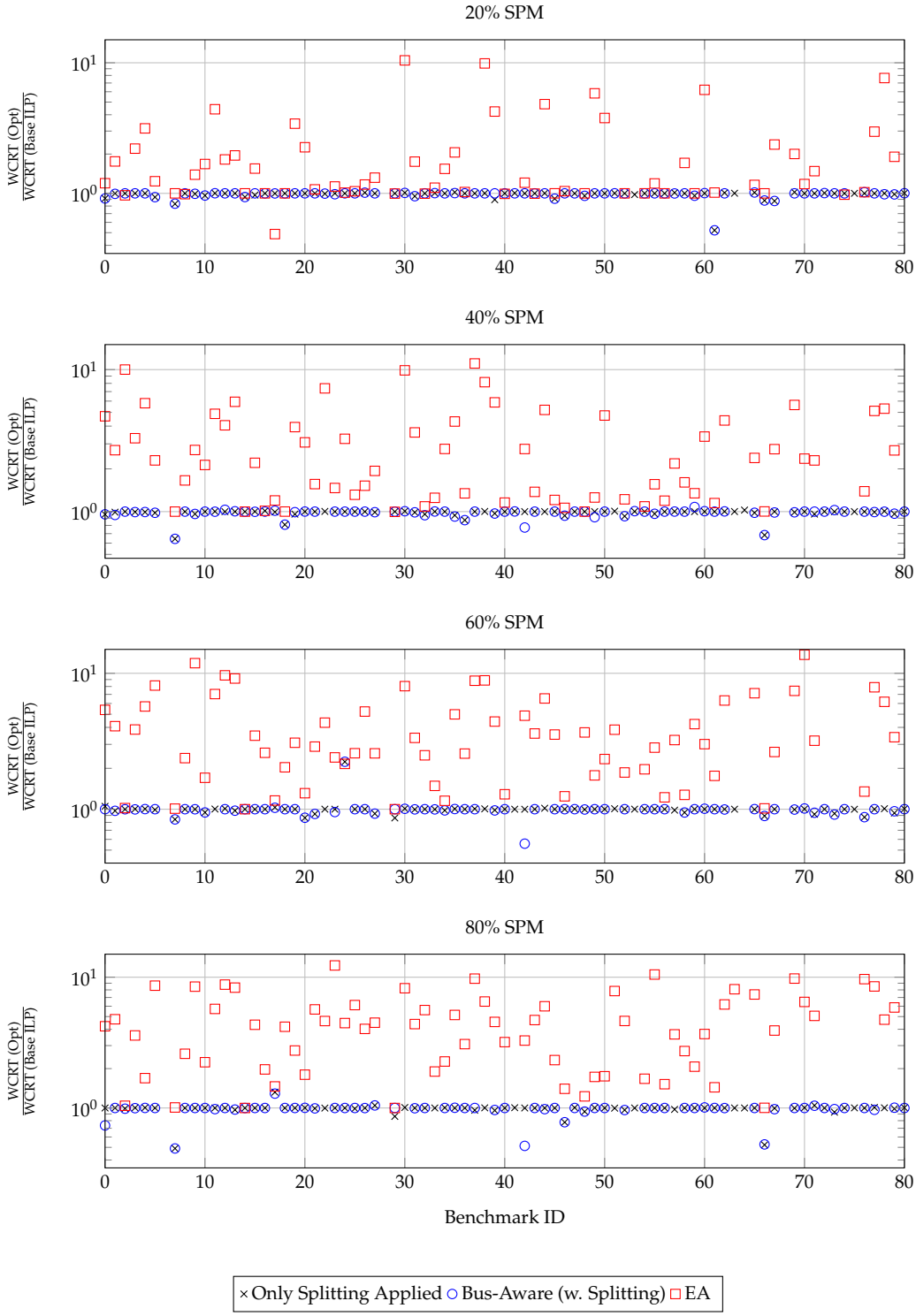


Figure 4.15. – WCRT change per benchmark when compared against the resulting WCRT of the reference ILP model for a system with 8 cores. Each graph represents an SPM configuration.

smallest WCET reduction by increasing the granularity for the optimization using sub basic block splitting occurs at a relative SPM size of 60%. Similarly, the average WCET improvement due to sub basic block splitting at a relative SPM size of 80% is significantly higher than compared to the results of the 60% SPM setting. This goes in line with the observations made for the multi-core systems and the bus-aware ILP-based optimization. It seems that on average, refining the basic block granularity is least effective in terms of WCET reduction at a relative SPM size of 60% to 70%. A possible explanation could be a similar program structure and amount of data access, as well as their distribution, for many benchmarks evaluated. This could lead to similar behavior among the benchmarks, as how large the SPM needs to be, such that the SPM allocation can take advantage of some basic blocks being refined.

In some particular cases, the bus-aware model is able to reduce the WCRT significantly. The largest improvement can be seen at an SPM configuration of 80% where the bus-aware ILP model reduces the WCRT by more than 50% for some cases compared to the base ILP model. For the other evaluated SPM configurations, the maximum improvements are mostly in a similar range with 47.9% at 20% SPM, 35.7% at 40% SPM and 44.2% WCRT reduction at an SPM configuration of 60%.

Similar as for the previous multi-core system evaluations, Figure 4.16 displays a subset of the evaluated benchmarks to highlight the potential of the bus-aware constraints. For the 20% SPM configuration, only very few and minor improvements can be seen compared to the basic block splitting applied alone. For the larger SPM configurations, more and greater WCRT improvements due to the bus-related constraints are shown. Similar as seen for the two- and four-core systems, the maximum improvement over the bus-unaware base model with basic block splitting is increasing with a larger SPM. For the 40% SPM configuration, an improvement of up to 22.7% can be seen, for 60% SPM up to 44.1% and for the 80% SPM up to 48.7% compared to the basic block splitting let alone.

4.3.5 Runtime

In this section, the required runtimes of the presented optimizations are evaluated. Figure 4.17 shows the average (arithmetic mean) runtime required to compile and optimize a single benchmark. As previously mentioned, a single core inside the assumed TDMA-based multi-core system can be analyzed and optimized in a fully isolated manner, as the allocation decisions of one core do not influence the timing of another core. Therefore, the required runtime is depicted on a single benchmark-basis in Figure 4.17 and is not distinguished for dual-, quad- or octa-core systems. Due to the timing isolation of each core, the overall runtime for a multi-core system scales linearly with the number of cores. The x-axis shows the different SPM sizes evaluated, whereas the y-axis depicts the required runtime in seconds.

As expected, the evolutionary approach on average takes the longest to finish with 1 145 s, as in most cases, the maximum number of generations cannot be reached within the time limit. Hence, the evolutionary approach will evaluate as many generations as possible within the given time limit and then return the best solution, leading to an average runtime close to the time limit. The average runtime of the evolutionary approach slightly decreases with an increasing SPM size. The reason for this lies in the repair function of an individual of the EA. In case the memory allocation resulting from a crossover or a mutation is invalid (due to too many basic

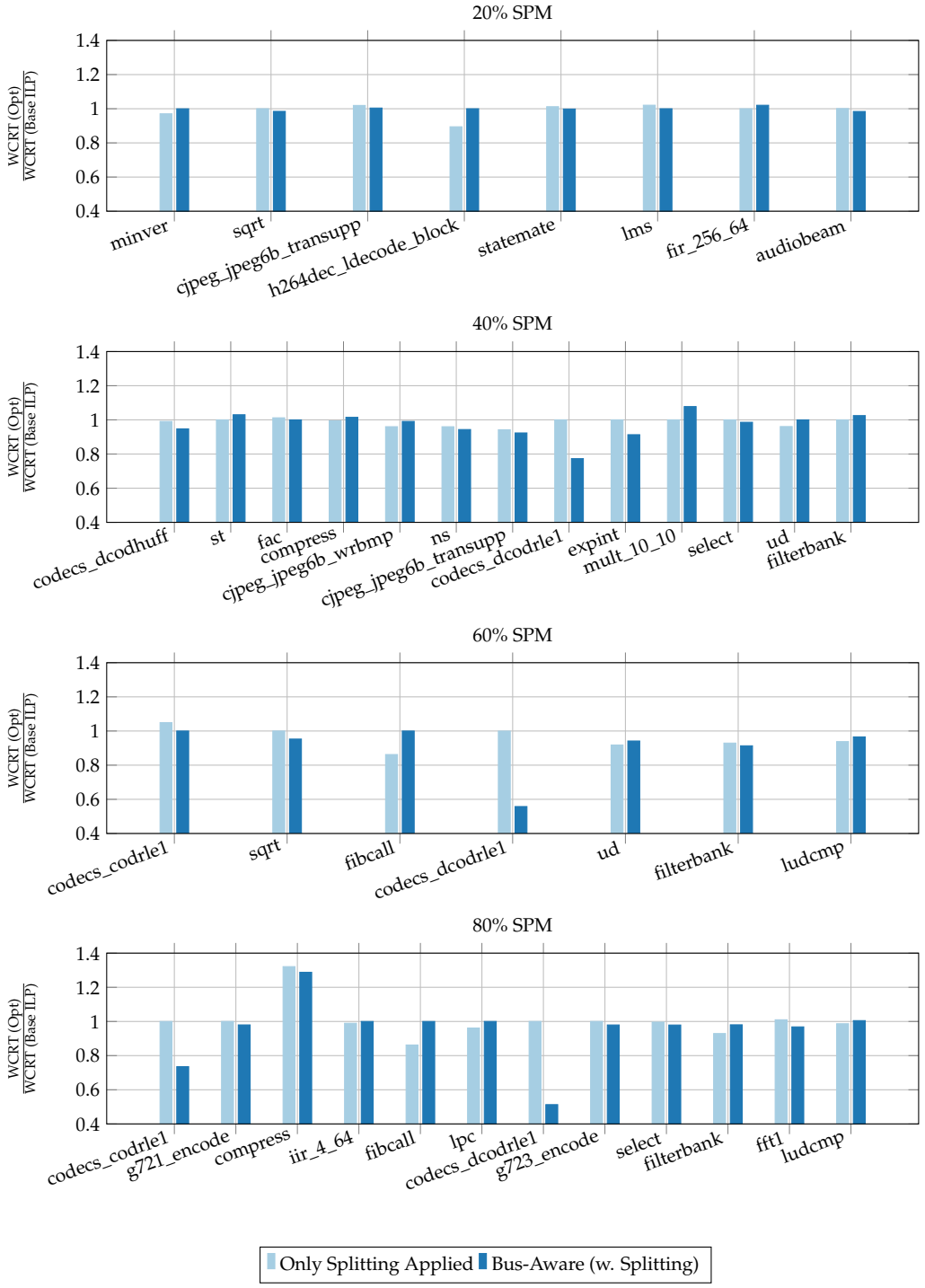


Figure 4.16. – WCRT change per benchmark when compared against the resulting WCRT of the reference ILP model for a system with 8 cores. A closer look at benchmarks whose WCRT improvement by only applying basic block splitting vs. further adding bus-timing constraints differ by more than 1% (ILP-based approaches only).

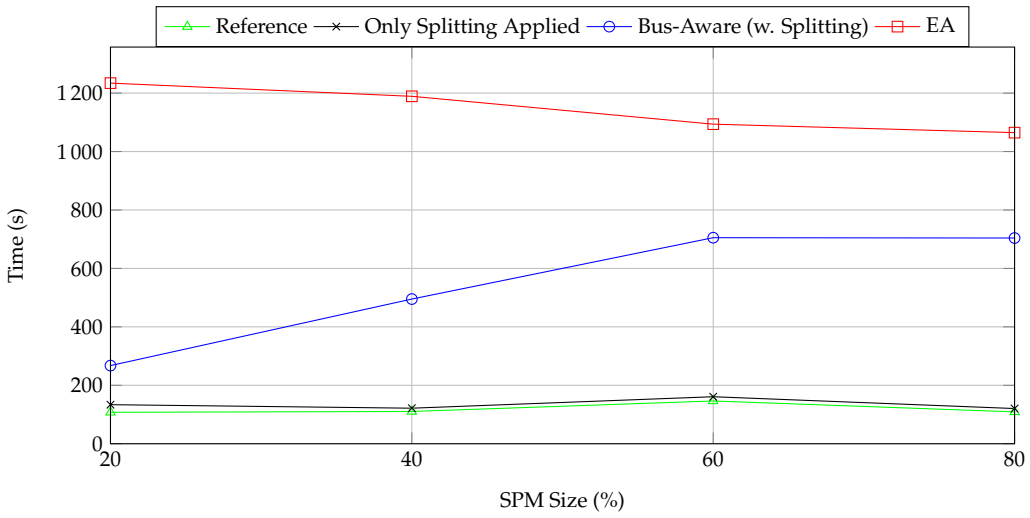


Figure 4.17. – Average runtime required for a single benchmark dependent on the SPM size.

blocks placed in the SPM), the corresponding individual is not discarded but repaired. During this repair process random basic blocks are iteratively removed from the SPM until the SPM is not overflowing anymore. A new memory allocation may lead to a broken control-flow graph, hence jumps need to be potentially fixed. Since the additional jump correction code can again *increase* the code size allocated in the SPM, the process of removing basic blocks and performing jump correction is done iteratively. With an increasing size of the SPM, a new memory allocation becomes less likely to be invalid, as the chance of the SPM being overfull is lower. Therefore, the average number of times the repair function needs to be called decreases with an increasing SPM size. This leads to the slight decrease of the average runtime of the evolutionary approach with a growing SPM.

In contrast, the approach based on the base ILP model always requires the least amount time. While the ILP-based approach only requires two timing analyses in total to create the ILP model, the EA requires one for *every* individual. On average, the base ILP model approach takes only 118 s for the entire compilation and optimization process. For the sake of completeness, the required runtime for the base ILP model with basic block splitting applied is shown as well. On average, this approach takes 134 s. This increase over the pure base ILP model can be easily explained due the additional constraints and variables added to the ILP due to the increased number of basic blocks. Finally, the fully bus-aware ILP approach requires on average 543 s. This drastic increase in runtime over the base ILP model is due to the additional number of constraints and variables added to the ILP to derive bus offsets and bus-related timings.

This increase in required runtime leads to a larger number of evaluations canceled due timeout. A closer look at this is shown in Figure 4.18. The different SPM settings evaluated are labeled on the x-axis, whereas the y-axis denotes the total number of canceled evaluations due to timeouts. The decreasing number of timeouts for the evolutionary approach with an increasing SPM size reflects the behavior already seen for the average runtime. Here, the decreasing number of required repair processes also leads to a lower number of timeouts.

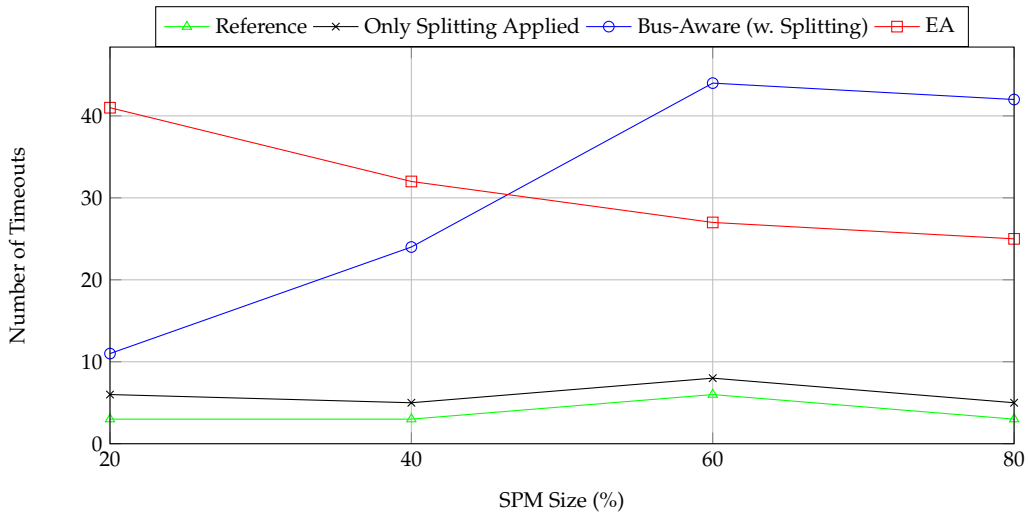


Figure 4.18. – Number of evaluations canceled due to timeouts dependent on the SPM size.

As expected, the approach based on the base ILP experiences the least amount of timeouts, while the base ILP model with basic block splitting slightly increases the number of timeouts. The number of evaluations canceled due to timeouts increases for the fully bus-aware approach when reaching larger SPM settings. At a small SPM setting of 20%, around 3.6 times more timeout occur for the bus-aware approach compared to the base ILP. For a larger SPM setting such as 80%, the number of timeouts at the bus-aware approach is 14 times higher than for the base ILP. The larger number of timeouts for an increasing SPM size can be explained due to the larger pool of potential solutions. For a small SPM, only very few basic blocks may fit into the SPM without violating the constraints limiting its size, reducing the overall complexity of the problem.

4.3.6 Conclusion

The evaluation of the presented ILP-based bus-aware instruction SPM allocation for TDMA-based multi-core systems leads to several interesting insights and conclusions. Overall, it is shown that for the majority of cases, the ideal SPM allocation in a TDMA-based multi-core system in regard to a minimal WCRT seems to be identical to the one for a simple single-core system. Yet for specific SPM size and program combinations, considering the bus-related timings during the SPM allocation can lead to significant further WCRT reductions. Using the proposed bus-aware ILP model, WCRT reductions of up to 47% compared against the already optimized WCRT of the base ILP model can be achieved.

The evaluation results also show that the influence of bus-related timings onto the optimal SPM allocation seems to increase with an increasing SPM size and number of cores in the system. The link between SPM size and importance of bus-related timings can be explained due to the larger number of basic blocks to be assigned to the SPM. In case of a comparably small SPM, only a few basic blocks can be assigned to it. These few basic block will most likely have already a large impact on the program's WCRT, as they will be in most cases part of a major loop of the program. Hence, even

if the ILP does not consider bus-related timings, it is likely that assigning these basic blocks into the SPM will lead to the minimal WCRT. With an increasing SPM size, the number of potential basic blocks to allocate increases as well. In most cases, not all basic blocks of a program will be part of a heavily nested loop, hence the average potential timing gain of a basic block will decrease with a larger SPM. Considering the bus-related timings can become crucial in these cases, as the estimated timing gain might become completely wrong otherwise. The trend of an increasing importance of considering bus-related timings during the SPM allocation with an increasing number of cores inside the system can be explained due to the increasing penalty of a false estimation. As the TDMA period increases with an increasing number of cores in the architecture assumed, so does the worst-case access time to the shared memory.

Furthermore, the evaluation results indicate that considering bus-related timings during an instruction SPM allocation has a greater influence on programs which are very data-focused. A large number of the benchmarks with an improved WCRT due to the additional bus-related constraints are focusing on processing data, such as encoding programs or programs working on global arrays in general. As all data objects are assumed to be placed in the shared memory (although the optimization allows data objects to be placed in either memory), the bus-aware ILP model can predict the bus offsets for these memory accesses. While a few benchmarks show a constant WCRT improvement due to the bus-aware ILP model, independent from number of cores or SPM size, in most cases only certain combinations lead to a significant further improvement. This can be related to the initial motivational example (cf. Section 4.2.3, Page 40), where a slight modification of the timings or the SPM size will change the importance of considering the bus-related timings.

While this evaluation showed the potential of taking bus-related timings into account during the optimization of a real-time multi-core system, it also revealed conflicting aspects. As the influence of bus-related timings onto the minimal WCRT increases with a larger SPM and number of cores, so does the required solving time. In general, it is shown that although the bus-aware additions can significantly improve the worst-case timing of a program, they also significantly increase the required solving time.

Overall, the presented ILP-based bus-aware static instruction memory allocation is no "one fits all" or "one fixes all" solution. It has strict requirements on the bus arbitration (TDMA with equally sized minimal slot lengths) and does not improve the worst-case timing over the bus-unaware approach in most cases. Yet, given the assumption that the bus arbitration requirements fit and a task of a system initially violates its deadline, even with a WCET-oriented SPM allocation applied, the bus-aware optimization can potentially improve the worst-case timing significantly in some cases and repair the system without any changes in the hardware or re-design of the software. Although the number of systems where the bus-aware optimization makes a difference is low, it is an important insight that even for a very restrictive bus arbitration like TDMA, where an individual core's behavior does not influence the other cores at all, a bus-aware memory allocation *can* make a significant improvement. The presented optimization is less attractive for programs in a rapid prototyping stage due to the longer optimization runtimes on average, but rather an optimization at the final stage where it is crucial that all tasks meet their deadline and the runtime of the optimization is not important.

5 Abstract System Behavior Description

As seen in the previous Chapter 4, specific multi-core-aware WCET-oriented code-level optimizations can have a promising potential. While the previous chapter introduced a bus-aware static instruction memory allocation for multi-core systems with a TDMA bus arbitration, this combination is only a small part of potential optimization or bus arbitration techniques and their influences on multi-core systems to be investigated. Especially for work-conserving bus arbitration techniques, where the behavior of one core can directly influence another, optimization decisions can have stronger consequences, as they are no longer locally isolated. This also potentially opens new optimization strategies which can take advantage of this interplay between the cores' behaviors. Yet with a growing number of cores and more complex bus arbitration schemes which do not enforce a complete temporal isolation between the cores like TDMA, performing multi-core-aware optimizations or timing analyses on the most detailed level quickly becomes infeasible. Therefore, a more generic and scalable foundation for performing optimizations and analyses is needed, while still having an insight into the microarchitectural layer. This chapter introduces a method to derive more abstract descriptions of a system's behavior from a code-level which can be used as a base for multi-core-aware optimizations and timing analyses. This is a compromise between a completely abstract system level and a most detailed microarchitectural level, while changes on a code-level still have an influence on the optimizations and analyses. The upcoming model is step stone to develop novel optimizations and analyses for a variety of different multi-core architectures, which would otherwise be not feasible.

A so-called *joint* worst-case execution time analysis of a multi-core architecture as presented in Section 2.2.1 achieves the highest possible level of precision. Possible bus-related contentions can be examined down to the level which single instructions may be executed by the parallel cores at the same time and hence may cause a delay. Additionally, it is potentially able to derive safe WCET values also for architectures with so-called timing anomalies. As the bus-related effects are analyzed during the microarchitectural analysis, possible uncertainties can be taken into account here and all potential pipeline states can be explored. Theses two major advantages are typically countered with the downsides of being computationally very expensive and having a poor scalability. Especially in case the cores cannot be handled individually (e.g., in case of a fixed priority bus arbitration), the required analysis time tends to explode. In reality, this means even for a simple quad-core architecture with small tasks (source code of less than 100 statements) allocated to each core (one task per core), the analysis time already easily goes beyond two hours [Kel15].

Purely compositional timing analyses as introduced in Section 2.2.3 in contrast typically feature a comparably good scalability, as they partition the analyses. As the microarchitectural WCET analysis happens completely independently from the

analysis of shared resources and the derived delays, the required complexity is reduced immensely. Therefore, compositional timing analysis frameworks can analyze even vastly distributed systems in a reasonable time. Compositional timing analyses typically handle tasks as complete black boxes and reduce them to a description of a few parameters, such as their isolated WCET, activation pattern and their access patterns to a shared resource. The access patterns to shared resources are typically described by an abstract metric. These metrics are required to derive a safe bound on the additional delay induced by possible concurrent accesses. With a more detailed metric, tighter bounds can be derived, yet also the complexity of the analysis increases. On the other hand, a more compact metric will mostly lead to shorter analysis times, yet also higher pessimism. A commonly used metric to describe task activation patterns or accesses to a shared resource are so-called *event arrival functions*. An event arrival function does not describe the pattern of so-called events in the *absolute* time domain, but compresses them into a description of their behavior in a time interval. An *upper* event arrival function $\eta^+(\Delta t)$ describes the maximum number of events appearing in a certain time interval, whereas a *lower* event arrival $\eta^-(\Delta t)$ describes the minimum number of events. This enables a higher abstraction level while keeping the potential for a very fine-grained characterization.

As event arrival functions are used, e.g., to derive safe upper bounds on bus contention delays, the accurate description of these functions is essential to the integrity of the overall analysis approach. In case the event arrival function underestimates the maximum number of events, the whole timing analysis may return an unsafe result. If an event arrival function is too pessimistic on the other hand, the timing result may classify a task to potentially miss its deadline, although this case is impossible in reality. Yet, event arrival functions are often simply derived by a set of measurements [CSBF18], a set of design requirements [RSMN11] or are simply assumed to be known [DN12a].

In order to derive precise and safe event arrival functions, an extraction on assembly code-level is presented in the following. Beside being able to derive a very precise description of the function, this also enables changes at the code-level to have influence on a timing analysis at system-level or lower. While a system-level analysis can still be done in a partitioned manner, effects at the microarchitectural level can be propagated onto the system-level using these precise event arrival functions. Therefore, advantages of system-level and low-level microarchitectural analyses can be combined: As the effects of code-level changes can be propagated to the system-level analysis, also fine-grained system behavior aspects can then be considered in the system-level analysis, yet the analysis approach remains partitioned. Finally, this approach also enables optimization opportunities for multi-core systems at the code-level. By applying the idea of event arrival functions, a level of abstraction is introduced. While this inevitably increases the level of pessimism in contrast to methods working on the lowest level of granularity, it also makes the analysis and optimization of complex systems manageable.

In the following Section 5.1, event arrival functions will be discussed in a greater detail. Section 5.2 gives an overview of related work in the field of event arrival function extraction. The description of event arrival functions using integer linear programming is described in Section 5.3. Here, Section 5.3.1 discusses the basic model for an upper event arrival function and Section 5.3.2 presents the basic model for

a lower event arrival function. The following sections introduce how several parts or aspects of a program can be modeled: Section 5.3.3 discusses loops, Section 5.3.4 presents the handling of flow facts, Section 5.3.5 introduces function calls and in Section 5.3.6 the required additions for periodic tasks are shown. Ways how to further improve the accuracy of the model are presented in Section 5.3.7. How a complete event arrival function can be sampled efficiently or be safely approximated is presented in Section 5.4: Section 5.4.1 introduces a method to quickly approximate a complete event arrival function, whereas Section 5.4.2 presents an efficient routine to precisely extract it. The following Section 5.5 walks through the ILP description of an event arrival function for an exemplary program. Finally, the effects of the different levels of granularities are evaluated in Section 5.6.

5.1 Definition of Event Arrival Functions

Event arrival functions are used to describe complex features of a system. Depending on the actual definition of an event, event arrival functions can describe multiple characteristics of a system, such as the activation of tasks or a task's shared resource access behavior. The idea of describing these specific patterns by their upper or lower limit in the interval time domain originates from Cruz's work on network delay analysis [Cru91] and Gresser's event stream model [Gre93]. Event arrival functions also build the fundamental basis of the network calculus by Boudec [LB98, LBT01] and the derived real-time calculus [TCN00].

Definition 5.1 (Event)**.** *A single event of an event stream is an abstract resource demand request, e.g., a task activation inside a multi-task system or a bus request inside a multi-core system.*

A given cumulative request function $R(t)$ describes the sum of events seen in the time interval $[0, t)$ with $t \in \mathbb{N}_0$. Based on that, an upper event arrival function $\eta^+(\Delta t)$ and a lower event arrival function $\eta^-(\Delta t)$ are defined, describing the upper and lower bound of $R(t)$ in the interval time domain [LBT01, Wan06]:

Definition 5.2 (Event Arrival Functions)**.** *With $R(t)$ describing the number of events seen in the interval $[0, t)$, $t \in \mathbb{N}_0$, the upper event arrival function $\eta^+(\Delta t)$ and lower event arrival function $\eta^-(\Delta t)$ are defined as:*

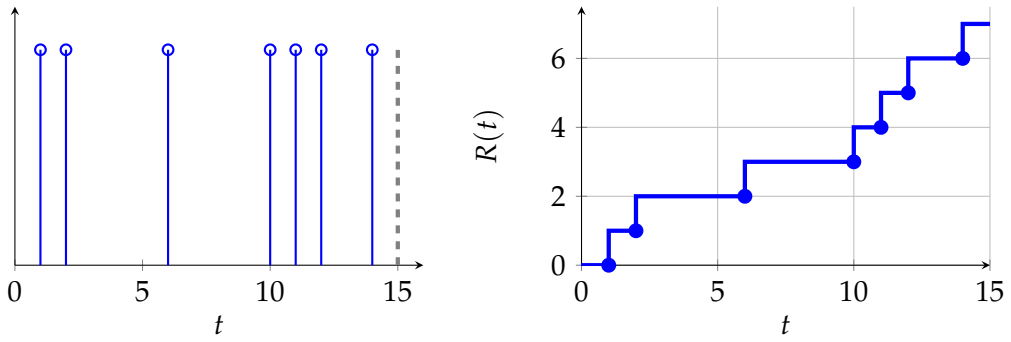
$$\eta^-(t-s) \leq R(t) - R(s) \leq \eta^+(t-s), \forall t \geq s \geq 0 \tag{5.1}$$

where $\eta^-(0) = \eta^+(0) = 0$ and $s, t \in \mathbb{N}_0$.

The upper event arrival function $\eta^+(\Delta t)$ is a cumulative function which describes an upper bound on the maximum number of events issued in *any* time interval of length of Δt. The lower event arrival function $\eta^-(\Delta t)$ is the corresponding counterpart, describing a lower bound on the minimum number of events issued in an interval of length Δt. Along the definitions of so-called "good" event arrival functions of Boudec [LBT01], event arrival functions should be either sub- or superadditive:

Definition 5.3 (Additivity of event arrival functions)**.** *A given upper event arrival function $\eta^+(\Delta t)$ should be subadditive:*

$$\eta^+(t+s) \leq \eta^+(t) + \eta^+(s), \forall t, s \geq 0 \tag{5.2}$$



(a) Exemplary bus access trace. **(b)** Cumulative request function.

Figure 5.1. – An exemplary bus access trace (a) and its corresponding cumulative request function $R(t)$ (b).

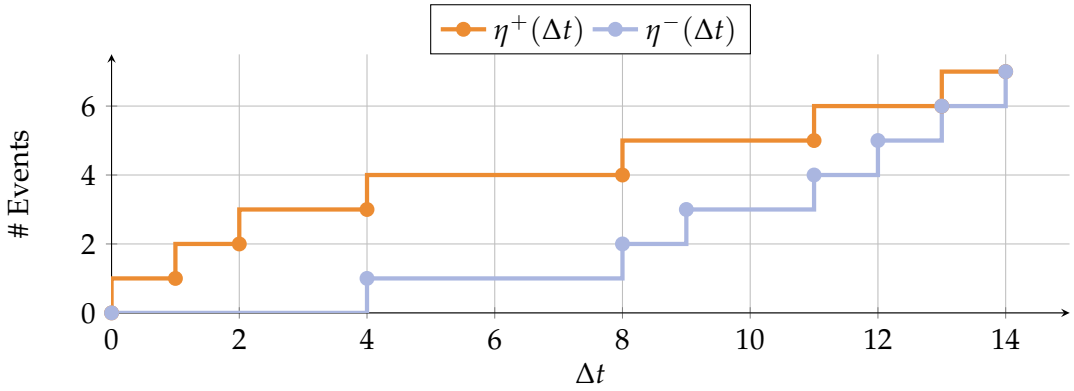


Figure 5.2. – Upper and lower event arrival functions $\eta^+(\Delta t)$ and $\eta^-(\Delta t)$ derived from the exemplary trace given in Figure 5.1.

Similarly, a lower event arrival function $\eta^-(\Delta t)$ is supposed to be superadditive:

$$\eta^-(t+s) \geq \eta^-(t) + \eta^-(s), \forall t, s \geq 0 \tag{5.3}$$

In case an upper (resp. lower) event arrival function is not subadditive (resp. superadditive), a tighter event arrival function can be derived by calculating its so-called subadditive (resp. superadditive) closure [Wan06]. An event arrival function without its corresponding additivity can, e.g., stem from defining an event arrival function solely based on given constraints of an event stream [LBT01].

In the following, a small example is given to illustrate the relationship between the cumulative request function $R(t)$ and its corresponding event arrival functions $\eta^-(\Delta t)$ and $\eta^+(\Delta t)$.

Example 5.1. *An exemplary trace of bus accesses is shown in Figure 5.1(a). This trace could have been generated by, e.g., snooping a bus for 15 time units. Each stem in the graph represents one bus access request. The corresponding cumulative function $R(t)$ is shown in Figure 5.1(b). The cumulative request function $R(t)$ equals the sum of all events that occurred in the interval $[0, t)$. Finally, the resulting event arrival functions $\eta^+(\Delta t)$ and $\eta^-(\Delta t)$ are shown in Figure 5.2. It should be noted that, in contrast to the previous figures, the Figure 5.2 is not displayed in an absolute time domain, but rather in an interval-domain. The upper event*

arrival function $\eta^+(\Delta t)$ displays an upper bound on the total number of bus accesses here in a given time interval, whereas $\eta^-(\Delta t)$ shows the lower bound. For example, when looking at an arbitrary time interval of 6 time units of the corresponding trace from Figure 5.1(a), this interval will have at most 4 accesses and least 1.

The previously mentioned additivity properties of event arrival functions can also be seen in Figure 5.2. For example, when regarding a time interval of length $\Delta t = 4$, $\eta^+(4) = 3$ is smaller than $\eta^+(3) + \eta^+(1) = 4$ or $\eta^+(2) + \eta^+(2) = 4$ due to the subadditivity of upper event arrival functions. Similarly, $\eta^-(4) = 1$ is greater than $\eta^-(3) + \eta^-(1) = 0$ or $\eta^-(2) + \eta^-(2) = 0$ due to the superadditivity of lower event arrival functions.

The previous example showcases how event arrival functions may be generated from captured traces. Yet, using traces to generate event arrival functions for the use in system-level analysis can be unsafe, as it is uncertain, if every possible situation is covered in the trace. This problem relates to the determination of a program's WCET by simply measuring its execution time several times and returning the highest. As one does not know if the worst case (or best case) was triggered during the observation, the resulting data is not guaranteed to be safe. Hence, any system-level analysis, which would use such data as an input, would also produce potentially unsafe results. It is therefore desirable to derive such event arrival functions in a safe fashion.

5.2 Related Work

While it is a common practice to use event arrival functions to describe the abstract demand requests of a task, a core or even a whole system in the domain of compositional timing analysis, the actual derivation of event arrival functions is less researched in detail. In the following, related approaches or commonly used assumptions are presented and discussed.

Jacobs et al. [JHH15] presented an approach to describe the upper event arrival function of a given task on a detailed low level using an ILP. The approach represents the program to be analyzed on a basic block level with known best-case execution times and maximum numbers of generated events per basic block. This work bases on the principles of the so-called IPET from Li and Malik [LM95]. As mentioned in Section 2.2.1, the IPET approach is typically used to determine the worst-case execution path of a program by describing the control-flow graph as a set of inequalities. Since the worst-case execution path of a program starts at its entrypoint and ends at a sink, the IPET enforces such a complete path. Jacobs et al. proposed a generalized IPET approach in which the path to be found is not restricted to start at the program's entrypoint and end at a sink, but can start and end at arbitrary basic blocks. In the following, a *complete* path through a program (starting at the entrypoint and ending at a sink) will be referred to as a *path*, while any other will be referred to as a *sub-path*. Instead of maximizing the execution time required to execute the chosen path, the number of generated events along the chosen sub-path is maximized. By limiting the execution time required to execute the sub-path to a given constant Δt, the ILP can then be solved to derive the value of the event arrival function $\eta^+(\Delta t)$. They also present how the model can be extended for strictly periodic programs. As this ILP model is used in a bigger context in the publication by Jacobs et al. [JHH15], the description of the required changes and additions to the original IPET are short and

informal. The following model proposed in this chapter of the presented thesis to describe an event arrival function of a given program follows the same principles as initially presented by Jacobs et al., yet covers all aspects by using a formal notion and extends these principles to also support lower event arrival functions. Furthermore, the precise handling of different loop types, function calls, flow facts and arbitrarily activated tasks are introduced, which the previous publication lacks of.

A different approach for deriving upper and lower event arrival functions for a program on a basic block level is presented by Schliecker et al. [SIE06]. The authors propose an explicit path searching algorithm to derive an event arrival function. By visiting every possible execution path of the program, an ordered list with the minimum (or maximum) time interval length between n events is generated. The authors also propose a method to merge event arrival functions of multiple tasks in the case of multi-task systems. It is not further discussed, how flow facts, function calls or periodic task activations can be expressed. The possibility of integrating such derived event arrival functions into compositional timing analysis tools is suggested by the authors [SNE09, SNE10, SSIE07] as well, although not actually evaluated.

Kleinsorge et al. [KFM13] presented a path analysis using explicit path enumeration as well. In contrast to "common" path analyses for the usage in worst-case timing analysis [AAN11, AMWH94], the presented approach focuses on not only deriving the worst-case execution path, but also the worst-case execution path for sub-paths. These sub-paths are not required to start at the program's entrypoint and end at a sink, but can start and end at arbitrary points.

A modified IPET model was presented by Altmeyer et al. [ABW09] in the context of a deferred preemption policy (also known as cooperative scheduling) [Bur95]. In a multi-task system with a deferred preemption policy, a task may not be preempted at any arbitrary point during its execution, but only at defined *preemption points*. The authors presented a modified IPET model which derives the longest possible execution time between two preemption points of the program. It differs from the ILP-based approach from Jacobs et al. [JHH15], as it is restricted to the execution time between two subsequent fixed points in the program under analysis.

Pellizzoni et al. [PSJ+10] presented an approach to derive an event arrival function from a given program and worst-case bus contention delays in a multi-core system. A single program is represented as a strictly sequential sequence of so-called *super blocks*. Each super block has a known execution time window and a known number of generated events when executed. As the sequence of super blocks is required to be linear, non-linear parts of the program under analysis (such as branches) need to be overapproximated by a single super block.

Beside deriving event arrival functions formally by means of a static program analysis, many works focus on approximating event arrival functions by using measurements or assuming a certain distribution. Carvajal et al. [CSBF18] presented a parallelizable algorithm to efficiently construct an event arrival function using measured traces. By executing the algorithm on a GPU, the proposed approach can quickly approximate an event arrival function from a set of measured traces, whereas the number of measured traces is in the magnitude of 10^6. A framework to analyze the bus contention in a multi-core system is presented by Dasari et al. [DNA15]. Here, the maximum number of bus accesses is assumed to be known and the actual distribution (corresponding to the event arrival function) is considered to be equidistant [DN12a].

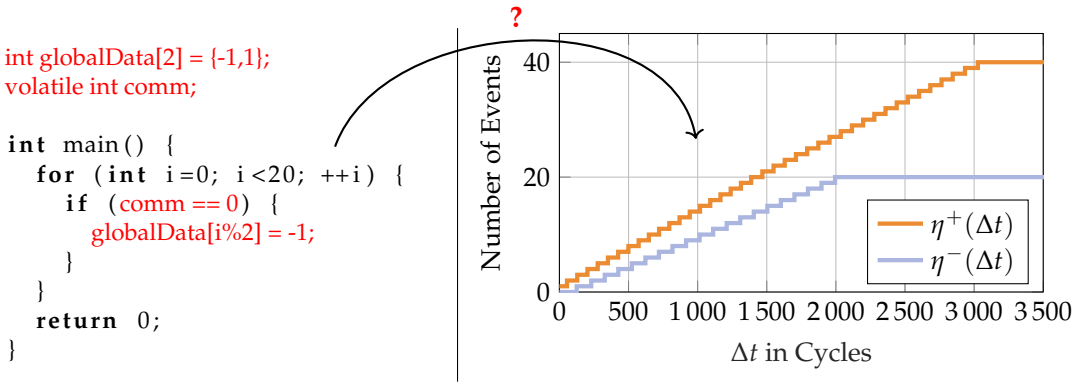


Figure 5.3. – Illustrative example of a program's given source code and the derived lower and upper event arrival functions from an assembly code-level.

The real-time calculus toolbox [WT06] offers a functionality to derive event arrival functions from a given set of traces, although the author warns that the resulting curves are not guaranteed to be safe, as no guarantees can be given on how complete the set of traces are [Wan06]. Salem et al. [SCF16] presented an approach to detect anomalies in a system by comparing a recent trace of events against the system's event arrival function. This "original" event arrival function is generated beforehand by using a set of measured traces.

5.3 Low-Level Event Arrival Curve Description

Instead of relying on captured traces to generate potentially unsafe event arrival functions, the core idea of the proposed approach is to use means of static analyses to derive these functions directly from the program. The following example illustrates the idea and the underlying problem.

Example 5.2. *In this example, the aim is to derive a safe upper and lower event arrival function on shared memory accesses initiated by a given program. On the left side of Figure 5.3, the program's source code is given. The two global variables are assumed to be allocated to shared memory and every read or write to them causes a shared memory access. A closer inspection of the source code gives clues about the absolute maximum and minimum accesses per program run. After a complete execution of the program, it is known that at least* 20 *accesses were performed (in case the "comm" variable is never equal to* 0*). It is also known, that at most* 40 *accesses were performed after a complete execution (in case the "comm" variable is always equal to* 0*). With this information from the source, the "heights" of the upper and lower event arrival functions after a complete program execution are known. Yet, the actual details of the event arrival functions, meaning how many accesses may occur in a specific time interval, cannot be derived from the source code. As accurate information on timings (e.g., number of cycles required for one loop iteration) can only be derived by knowing the architecture and machine instructions to be executed, an event arrival function extraction needs to be placed on the assembly code-level. By doing so, the event arrival functions on the right-hand side of Figure 5.3 for an exemplary architecture could be deduced. As these event arrival functions are generated by statically analyzing the program on a code-level, the resulting curves are safe and tight. In contrast to captured traces, the analysis is guaranteed to cover all possible paths.*

This is done by taking analyses known and heavily used in the domain of worst-case execution time analysis and adapting them. For this purpose, the event arrival function is described using an ILP. Section 5.3.1 presents the basic ILP which models an upper event arrival function of a basic program. The required re-formulations to describe a lower event arrival function are presented in Section 5.3.2. The handling of loops is described in Section 5.3.3, flow facts in Section 5.3.4 and function calls in Section 5.3.5. Section 5.3.6 introduces how the model can be extended to cover arbitrarily activated tasks. Further refinements to achieve a higher precision are presented in Section 5.3.7. With an extensive ILP description of an event arrival function present, the subsequent Section 5.4 showcases how an actual event arrival function can be derived from the ILP with varying granularities. A complete set of constraints for an exemplary control-flow graph is derived in Section 5.5. This chapter closes with Section 5.6 evaluating different parameters and their influence on a system-level timing analysis.

For the upcoming sections, the following assumptions are made:

- The control-flow graph on an assembly code-level of the program under analysis is known and given.
- The worst-case and best-case execution times on a basic block-level are known.
- The maximum and minimum number of events per single execution of each basic block is known.
- No out-of-order (OOO) execution. Although the model is also applicable to OOO architectures by transforming the control-flow graph into a corresponding analysis graph [Kel15], it is excluded due to the reasonably low popularity of OOO architectures in hard real-time systems.

The actual definition of an event is held abstract in the rest of this chapter on purpose. While in the context of this thesis, an event will be regarded in most times as a bus request, the general model is widely applicable. E.g., the call of a certain function can be defined as event. This way, the activation pattern of this specific function can be deduced. An event could also be defined as a write-action to a specific memory address. As, e.g., sending messages over a field area network (e.g., CAN) is often triggered by writing to a memory-mapped register, this could be defined as an event. Furthermore, also implicit memory accesses, such as fetching instructions from a shared memory, can be defined as an event.

The idea of using an adapted version of the IPET to derive an upper event arrival function was first presented by Jacobs et al. at the *International Conference on Real Time and Networks Systems 2015* [JHH15]. This includes a short, mostly informal description of how an upper event arrival function of a basic program can be modeled using an ILP. Similar to the idea of a "generalized IPET" by Jacobs et al., the author of this thesis presented a detailed model for deriving event arrival functions at the *Euromicro Conference on Real-Time Systems (ECRTS)* in Spain 2018 [OSF18] with the following novel contributions: 1) Description of *lower* event arrival functions, 2) precise handling of different loop types, 3) precise handling of function calls and 4) extraction of complete event arrival functions with an adjustable granularity. This thesis further introduces the handling of so-called flow facts and arbitrary activation patterns for periodic tasks.

5.3.1 Upper Event Arrival Function

As previously discussed, an upper event arrival function $\eta^+(\Delta t)$ describes the maximum number of events occurring in a time interval of length Δt. Seen from a program point-of-view, a single event (e.g., a shared memory access) is typically generated by executing a certain instruction (e.g., a load-instruction). Therefore, for any given Δt, $\eta^+(\Delta t)$ is defined by a *sub-path* in a given program over which the maximum number of events can be generated, yet takes at most Δt time units to execute. A sub-path may start and end at arbitrary basic blocks of a program and may even span multiple instances of the said program.

A path analysis technique well-known in the domain of worst-case execution time analysis is the so-called Implicit Path Enumeration Technique (IPET) by Li and Malik [LM95] (see also "path analysis" in Section 2.2.1). As the name states, it does not require to search over every possible path *explicitly*, but the set of all paths is described implicitly. The aim is to find that path along which the longest execution time of a program occurs, given a control-flow graph with known worst-case execution times of each basic block. By utilizing the IPET, the control-flow graph is described as a set of integer linear equations and an objective is set to find the path through the program, which requires the longest time. A key feature of this approach is that so-called flow facts can easily be integrated as additional constraints. Thereby, maximum recursion depths or nested triangular loops can be integrated into the ILP model in a simple fashion.

The general idea of the IPET to model all possible paths of a program using a set of integer linear equations is used here as well, yet adapted to not only find a complete path through the program, but also sub-paths. The timings of a basic block are considered in terms of CPU cycles, such that they can be represented using integers. As preparation step, the control-flow graph is synthetically modified by inserting a virtual source block. This block only exists in the control-flow graph used for the analysis, the actual program code remains untouched. A virtual source $\ominus$ is inserted as a predecessor of the very first basic block of the program. Subsequently, a basic control-flow restricting constraint is generated for each block i:

$$\left(\sum_{j \in \mathcal{P}_i} p_{j,i}\right) - e_i = \sum_{k \in \mathcal{S}_i} (p_{i,k} - s_{i,k}) \tag{5.4}$$

The set $\mathcal{P}_i$ contains all direct predecessors of basic block i, whereas the set $\mathcal{S}_i$ holds all direct successors. The ILP variable $p_{j,i}$ represents the number of times the control-flow edge from basic block j to basic block i is executed. The number of times a basic block is executed is defined by the total number of executions along all incoming edges. e_i is an ILP variable restricted to binary values, representing whether basic block i is the last block ($e_i = 1$) or not ($e_i = 0$) of the sub-path to find. In analogy to this, $s_{i,k}$ is an ILP variable limited to binary values as well, denoting whether the control-flow edge from block i to k is used as the start of the sub-path ($s_{i,k} = 1$) or not ($s_{i,k} = 0$). The differentiation *which* incoming control-flow edge of a basic block starts the sub-path is later used to handle loops in the model. Equation (5.4) enforces the basic structure of the control-flow graph and resembles Kirchhoff's point law. Simply spoken, if the control-flow arrives x times a basic block i, it also has to leave

x times. The only exception to this balance is granted when the chosen sub-path should end or start at this block. If the sub-path *ends* at basic block i, e_i is set to 1 and effectively removes one flow (a single execution of an edge) from the left-hand side of the equation. Similarly, if the edge from basic block i to its successor k is used as the start of the sub-path, one flow is removed from the right-hand side of Equation (5.4).

As only a single sub-path should be determined, the sum of all variables representing the start of a sub-path is set to be 1. This is done as well for all variables representing the end of a sub-path. For a sub-path exactly one start and one end point have to exist, therefore the sum over all starting point variables $s_{i,j}$ has to be equal to the sum over all ending point variables e_i:

$$\sum_{i \in \mathcal{B}} \sum_{j \in \mathcal{P}_i} s_{i,j} = \sum_{i \in \mathcal{B}} e_i = 1 \tag{5.5}$$

The set $\mathcal{B}$ contains all basic blocks of the current program. Furthermore, a control-flow edge from block i to j can only be selected as a starting edge, if this edge is also executed at least once. Therefore, the following constraint is inserted for each control-flow edge:

$$s_{i,j} \leq p_{i,j} \tag{5.6}$$

For the sake of an easier notation, a helper variable s_i is introduced, representing that basic block i is used as the starting basic block of the sub-path.

$$s_i = \sum_{j \in \mathcal{P}_i} s_{j,i} \tag{5.7}$$

Since an incoming edge represents the execution of a basic block, s_i is defined by the sum of all incoming start edge indicator variables. Hence, if any edge to basic block i is used as a starting edge, it represents that the sub-path starts at basic block i.

An additional ILP variable a_i^+ represents the maximum number of events contributed along the sub-path by basic block i, whereas the overall number of events along the sub-path is denoted as a_{Total}^+. The maximum number of events generated by a single execution of basic block i is denoted as the constant A_i^+. This constant has to be derived beforehand. In case an event represents a shared memory access, this can be done by, e.g., performing a value analysis to identify the potential accesses.

$$a_i^+ \leq A_i^+ \cdot \sum_{j \in \mathcal{P}_i} p_{j,i} \tag{5.8}$$

$$a_{\text{Total}}^+ = \sum_{i \in \mathcal{B}} a_i^+ \tag{5.9}$$

The sum in Equation (5.8) describes the execution count of basic block i along the chosen sub-path (note that the subtraction of the sub-path end indicator variable e_i in Equation (5.4) does not reduce the actual execution count, but simply preserves the validity of the point law equation). The relational operator in Equation (5.8) is not an equality- but a lower than or equal to-operator to enable the extraction of an event arrival function for different granularities later on.

With a^+_{Total}, the number of events along the sub-path is known, yet the timing is still unbounded. This is resolved using the variable z^+_i for each basic block i. z^+_i denotes the minimum number of cycles required for executing basic block i on the sub-path.[1]

$$z^+_i \geq \left(C^-_i \cdot \sum_{j \in \mathcal{P}_i} p_{j,i} \right) - \left(C^-_i - 1 \right) \cdot b_i \tag{5.10}$$

C^-_i denotes the best-case execution time of basic block i. As the aim is to find a sub-path along which the *maximum* number of events is generated while taking at most Δt cycles, the shortest possible execution time of each basic block has to be used here. b_i is a reduction factor and can be either 0, 1 or 2. Its purpose is to cover corner cases, when the sub-path starts or ends at basic block i and is defined as follows:

$$b_i = \begin{cases} 0 & \text{if } s_i = e_i = 0, \\ 2 & \text{else if } s_i \wedge e_i \wedge \left(\sum_{j \in \mathcal{P}_i} p_{j,i} > 1 \right), \\ 1 & \text{else.} \end{cases} \tag{5.11}$$

The need to potentially lower the number of cycles contributed by a basic block i to the sub-path's required time to execute is owed to the granularity of basic blocks. As a basic block is simply represented by its maximum number of generated events A^+_i and its best-case execution time C^-_i, no deeper information about the actual distribution of the events during its execution is taken into account. Though the ILP model represents the start and end point of the sub-path on the granularity of basic blocks, this does not need to be the case for an actual sub-path leading to the maximum number of events in a given time interval. E.g., when regarding two adjacent basic blocks, one with an event-triggering instruction at the very end and the next block with one event-triggering instruction at the very beginning, the time required to cover those two events will be far less than the sum of both basic blocks' timings. As safe overapproximation, it is assumed that all events are generated in the very *last* cycle of a basic block in case this block is used as a starting point. Similarly, it is assumed that all events are generated in the very *first* cycle of a basic block in case it is used as an ending point. In both cases, $C^-_i - 1$ clock cycles are subtracted from the execution time contributed by the basic block i along the sub-path, assuming it only takes a single cycle ($C^-_i - C^-_i + 1$) to trigger the events of the basic block. This single cycle is simply a safe assumption for a minimum time required to trigger the events of the basic block. Methods to further tighten these approximations are presented in the upcoming sections. This reduction of the timing is required, as otherwise the full best-case execution time C^-_i would be assumed for the sub-path, even though, e.g., the sub-path starts at the last instruction of a large basic block which triggers an event (which could lead to an unsafe upper event arrival function $\eta^+(\Delta t)$, as potentially more events could be generated in a time interval).

In case the basic block i is neither used as the starting or ending point of the sub-path, no timing reduction is required and hence b_i is set to 0. If basic block i is used as

[1] While the superscript $^+$ typically indicates a *maximum* timing in this thesis, it is used here to indicate that the variable is part of the ILP model for an *upper* event arrival function $\eta^+(\Delta t)$.

the start *and* end of the sub-path and is executed more than once, the timing reduction factor b_i is set to 2. That a basic block i is chosen as a start and end point of a sub-path can result from two possibilities:

1. The sub-path only consists of basic block i. In this case, basic block i is only executed once and the timing should only be reduced by $\left(C_i^- - 1\right)$.
2. The basic block is part of a loop, whereas the start and end point of the sub-path lie in different loop iterations. In this case, the timing should be reduced by $2 \cdot \left(C_i^- - 1\right)$.

The last case of Equation (5.11) covers that a basic block is either the start or end point, or the sub-path only consists of basic block i. In these cases, the timing reduction factor is set to 1.

If given a constant time interval length Δt, the maximum number of events occurring in this interval can then be determined by setting the following constraint and objective:

$$\Delta t \geq \sum_{i \in \mathcal{B}} z_i^+ \tag{5.12}$$

$$\max : a_{\text{Total}}^+ \tag{5.13}$$

5.3.2 Lower Event Arrival Function

Beside deriving the upper event arrival function $\eta^+(\Delta t)$ of a given program, the lower event arrival function $\eta^-(\Delta t)$ can be extracted as well. The lower event arrival function $\eta^-(\Delta t)$ represents the minimal number of events that are generated in a time interval of length Δt. Most concepts of the previously introduced ILP model can be directly adopted, yet a few aspects require modifications. In the following, the required changes and additions to the base model are discussed.

The basic point laws of each basic block as introduced in Equation (5.4) can be directly adopted. The inequation limiting the number of events generated by a basic block i on the sub-path (cf. Equation (5.8)) is replaced by the following constraint:

$$a_i^- \geq \left(A_i^- \cdot \sum_{j \in \mathcal{P}_i} p_{j,i} \right) - b_i \cdot \left((s_i \wedge u_s) \vee (e_i \wedge u_e) \right) \cdot A_i^- \tag{5.14}$$

A_i^- represents the minimum number of events generated by a single execution of basic block i. In contrast to the model deriving the upper event arrival function, Equation (5.14) also contains a subtractive term. The reasoning behind this is analogous to the subtractive term for limiting the accumulated execution time of a basic block in Equation (5.10): As the distribution of events inside the basic block i is handled as a black box, basic block i may be partially executed without generating any of the A_i^- events. As a safe approximation, it is assumed that in case the sub-path starts at basic block i, all events are generated in basic block i's very first execution cycle. Similarly, if the sub-path ends at basic block i, it is assumed, that all events are generated in basic block i's very last execution cycle. This reduction is required for a safe lower event arrival function. For example, a sub-path without any events may start at a basic block i, although basic block i contains an event-triggering instruction (here, the sub-path

would simply start after this instruction). The reduction term in Equation (5.14) covers this case, as it enables the sub-path to start somewhere in basic block i without the events being accounted for (as the exact distribution of event-triggering instruction inside a basic block is not reflected in the model).

Whether such a reduction in events is applied at the starting or ending block of the sub-path is represented by the corresponding binary variable u. If u_s is set to 1, the number of events generated by the starting block of the sub-path i is reduced by $b_i \cdot A_i^-$. The same applies for the ending block of the sub-path and the corresponding variable u_e. Note that b_i is determined identically as previously as presented in Equation (5.11). The variables u_s and u_e only exist each once in the ILP model as they represent whether a reduction of events is applied to the start (or respectively to the end) of the sub-path. In case u_s and u_e are set to 0 by the ILP solver, no reductions in terms of events or timing are applied to the boundary blocks of the sub-path. While the subtractive term of Equation (5.14) may appear non-linear due to its multiplication, it can easily be expressed by a simple case-structure, as the term $((s_i \wedge u_s) \vee (e_i \wedge u_e))$ is bound to binary values and A_i^- is constant.

The constraint for the minimum number of events along the sub-path is very similar to the Equation (5.9) for the model of an upper event arrival function:

$$a^-_{\text{Total}} = \sum_{i \in \mathcal{B}} a_i^- \tag{5.15}$$

Following, the number of accumulated execution cycles per basic block i (cf. Equation (5.10)) is replaced by the following inequation:

$$z_i^- \leq \left(C_i^+ \cdot \sum_{j \in \mathcal{P}_i} p_{j,i} \right) - b_i \cdot ((s_i \wedge u_s) \vee (e_i \wedge u_e)) \tag{5.16}$$

C_i^+ is the worst-case execution time of a single execution of basic block i. When deriving the lower event arrival function $\eta^-(\Delta t)$, the WCET of a basic block instead of its BCET is chosen, as $\eta^-(\Delta t)$ is determined by the *least* number of events inside a time interval of length Δt. Therefore, the WCET per basic block has to be used, as this ensures the *maximum* time between events. In contrast to previous constraints limiting the accumulated execution time per basic block (cf. Equation (5.10)), the subtractive term of Equation (5.16) may only reduce z_i^- by 1 or 2 cycles (as $b_i \in \{0, 1, 2\}$). This goes along with the previous Equation (5.14): If basic block i is used as the starting block of the sub-path ($s_i = 1$) and not the ending block ($e_i = 0$), b_i is set to 1. Hence, the required execution time can be reduced by 1 if u_i is set to 1. This relates to the intention, that all events of the block i are assumed to be executed in the very *first* cycle of i and the sub-path only starts from the *second* cycle of the basic block. Hence, the number of events generated by basic block is reduced by A_i^- by Equation (5.14) and the execution time is lowered by a cycle. The cases of basic block i being the ending block or the start and end follow analogously.

The WCET per basic block used in Equation (5.16) has to be derived carefully. While the BCET can typically be safely derived in an isolated fashion (i.e., with no interference from other cores or tasks), the WCET of a basic block is easily influenced by concurrent cores or tasks. In case the event type influences the WCET (e.g., shared

memory access), a possible method is to first derive the upper event arrival function $\eta^+(\Delta t)$ and use this to derive safe WCETs using, e.g., system-level analyses.

Additionally, a variable f indicating whether a *full* path of the program is taken (i.e., the sub-path starts at the program's entrypoint and ends at an exit) is created. As helping variables, $s_\ominus$ and $e_\perp$ are introduced, representing whether the sub-path starts at the program's entrypoint ($s_\ominus = 1$) or ends a program's exit ($e_\perp = 1$).

$$s_\ominus = s_{\ominus,j} \tag{5.17}$$

$$e_\perp = \sum_{i \in \mathcal{T}} e_i \tag{5.18}$$

$$f = s_\ominus \wedge e_\perp \tag{5.19}$$

The basic block j in Equation (5.17) is assumed to be the program's entrypoint. The set $\mathcal{T}$ contains all exiting blocks of the program.

Finally, the previous constraint limiting the total accumulated execution time along the sub-path (cf. Equation (5.12)) and the actual objective term from Equation (5.13) are replaced by the following ones:

$$\Delta t \leq \left(\sum_{i \in \mathcal{B}} z_i^- \right) + (f \wedge \overline{(u_s \vee u_e)}) \cdot M \tag{5.20}$$

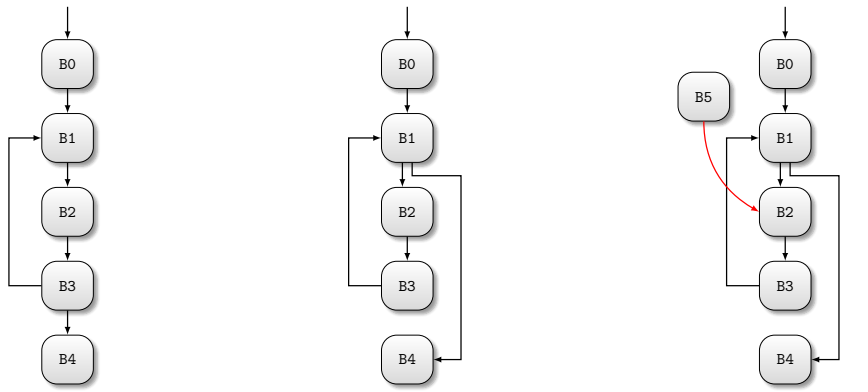
$$\min : a^-_{\text{Total}} \tag{5.21}$$

Equation (5.20) ensures that the accumulated execution time along the sub-path is greater than or equal to the given time interval length Δt. M is a sufficiently large enough constant. In this case, a trivial value for M is Δt. The relational operator of Equation (5.20) is flipped in comparison to Equation (5.12) of the upper event arrival function model, as now the objective function is set to *minimize* the number of events along the sub-path. Hence, the solver is forced to find a sub-path, which takes at least Δt cycles while minimizing the number of events along. Additionally, Equation (5.20) is always fulfilled, if a full path through the program is chosen and no reduction in terms timing or events is applied to the start ($u_s = 0$) and end block ($u_e = 0$) of the path. While f equals 1 when the entrypoint of the program is used as a start of the sub-path and an exit block as the end of the sub-path, this may also describe a sub-path which starts and ends somewhere *inside* these basic blocks and does not necessarily include the entire starting and ending blocks. The sub-path is therefore a complete path through the program (fully including the start and end blocks) if $f \wedge \overline{(u_s \vee u_e)}$ is true.

This is required for the non-periodic base model, as otherwise the ILP would become infeasible for values of Δt larger than the program's WCET (since no sub-path longer than the program's WCEP exists). Hence, the lower event arrival function $\eta^-(\Delta t)$ converges to the minimal number of events generated by a complete execution of the program.

5.3.3 Loops

The previously introduced models do not limit loop iterations. Without limiting the maximum number of iterations per loop, the introduced model may return wrong



(a) Tail-controlled. **(b)** Head-controlled. **(c)** Head-controlled (irr.).

Figure 5.4. – Exemplary loop structures.

results, as events inside loops can be accumulated indefinitely. Therefore, loop structures need to be bounded by their so-called upper loop bound, representing the maximum number of times a loop may be executed. To further increase the tightness of the model, also the lower loop bound and the loop type are considered.

In the following, loops are classified as head- and tail-controlled loops. Figure 5.4(a) displays an example for a tail-controlled loop, whereas Figure 5.4(b) shows an exemplary head-controlled loop. The set containing all tail-controlled loops of a task is denoted as $\mathcal{L}_\mathrm{T}$. A set $\mathcal{N}_\ell$ contains all back-edges of a loop ℓ. A back-edge of a loop transfers the control to the next loop iteration (e.g., see edge `B3` $\rightarrow$ `B1` in Figure 5.4(a)).

In case of a tail-controlled loop, an additional constraint is generated to limit the flow through all back-edges:

$$\forall \ell \in \mathcal{L}_\mathrm{T} : \sum_{(i,j)\in\mathcal{N}_\ell} p_{i,j} \leq n_\ell^T \tag{5.22}$$

n_ℓ^T limits the total flow through all back-edges of the loop ℓ. For a tail-controlled loop, this is defined by the loop's upper bound $\mathrm{B}_\ell^{\mathrm{Up}}$, the number of flows into the loop and whether the sub-path starts inside the loop or not.

$$n_\ell^T = \left(\mathrm{B}_\ell^{\mathrm{Up}} - 1\right) \cdot \Bigg(\underbrace{\sum_{i\in\mathcal{E}_\ell} \sum_{j\in(\mathcal{P}_i\setminus\mathcal{M}_\ell)} p_{j,i}}_{\mathrm{I}} + s_\ell \Bigg) \tag{5.23}$$

The set $\mathcal{E}_\ell$ contains all basic blocks which are an entrance to the loop ℓ. A loop entrance of a loop ℓ is defined as a basic block which is part of the loop ℓ and has a preceding basic block, which is not part of the loop. For example in Figure 5.4(a), basic block `B1` is a loop entrance (and also the only one). The set $\mathcal{M}_\ell$ contains all basic blocks which are part of the loop ℓ (also nested loop members). Therefore, the term I describes all flows into the loop ℓ from outside the loop. s_ℓ is an ILP variable restricted to binary values. It represents whether the sub-path starts inside the loop ℓ ($s_\ell = 1$) or not ($s_\ell = 0$). A sub-path is also denoted to start in loop ℓ if it starts in a nested loop inside of the loop ℓ or in a function called inside the loop. In combination, Equation (5.22)

and Equation (5.23) allow the loop body to be executed up to B_ℓ^{Up} times for each time the loop is entered. Equation (5.23) uses the term $(B_\ell^{Up} - \mathbf{1})$, as a tail-controlled loop *always* executes the loop body at least once, hence the maximum execution count of the back-edges has to be reduced by one. Furthermore, the loop body may also be executed additionally up to B_ℓ^{Up} times if the sub-path has its starting point inside a loop iteration.

s_ℓ is set as follows:

$$s_\ell = \left(\bigvee_{i \in \mathcal{M}_\ell} \bigvee_{\gamma \in \mathcal{F}_i} s_\gamma \right) \vee \bigvee_{i \in \mathcal{M}_\ell} \begin{cases} \bigvee\limits_{j \in (\mathcal{P}_i \cap \mathcal{M}_\ell)} s_{j,i} & \text{if } i \in \mathcal{E}_\ell, \\ \bigvee\limits_{j \in \mathcal{P}_i} s_{j,i} & \text{else.} \end{cases} \tag{5.24}$$

There are two ways how a sub-path may start during the execution of a loop: Either at a basic block which is part of the loop (including potential nested loops), or inside a function called from within the loop (or functions called within this function). The second possibility is covered by the first term of Equation (5.24). The set $\mathcal{F}_i$ contains all potentially directly called functions from basic block i. Since the control-flow graph with all call edges and their potential targets is known, the set $\mathcal{F}_i$, as well as all functions called from within function, can easily be derived by analyzing the control-flow graph. The variable s_γ is set to 1 in case the starting point of the sub-path is placed inside function γ or in functions called by nested calls in this function. It is set using the following constraint:

$$s_\gamma = \bigvee_{i \in \mathcal{B}_\gamma} s_i \vee \bigvee_{\beta \in \mathcal{F}_i} s_\beta \tag{5.25}$$

$\mathcal{B}_\gamma$ contains all basic blocks belonging to function γ.

The case where a sub-path starts at a basic block within the loop ℓ is covered by the second term of Equation (5.24). A sub-path starts inside a loop ℓ, if a member basic block of this loop is used as a start, yet with an exception for the entrance blocks of a loop. In case a control-flow edge from outside the loop is used as a starting edge at an entrance block (e.g., `B0` $\rightarrow$ `B1` at Figure 5.4(a)), the upper loop bound is already set correctly (see Equation (5.23)). These edges are therefore excluded for the entrance blocks (first case of Equation (5.24)). The set $\mathcal{P}_i \cap \mathcal{M}_\ell$ contains all basic blocks which are a direct predecessor of basic block i and are also members of the loop ℓ. By performing a logical OR operation over all possible $s_{j,i}$ variables in Equation (5.24), it is determined whether the sub-path uses an edge inside the loop as a starting point or not.

Head-controlled loops are handled in a similar manner, yet with slight modifications. Figure 5.4(b) shows an exemplary head-controlled loop. For head-controlled loops, the number of flows into the loop body is restricted instead of the back-edges as described for tail-controlled loops. If head-controlled loops would be also handled by restricting the back-edges, one additional loop iteration would be permitted by the constraints than actual possible.

Example 5.3. *The head-controlled loop in Figure 5.4(b) is assumed to have an upper loop bound of 1, meaning the loop body is allowed to be executed at most 1 time. If the sub-path's starting point is chosen to be at basic block* `B1`*, the back-edge (*`B3` $\rightarrow$ `B1`*) is allowed to be*

executed at most 1 time, as no flow is entering the loop. Yet, considering the basic flow constraints introduced in the previous section, the sub-path $\{$`B1`, `B2`, `B3`, `B1`, `B2`, `B3`$\}$ *would be valid and does not violate the constraints, as the back-edge is only executed once. As this relates to the loop body being executed once more than actually allowed, just restricting the maximum flow through the back-edges does not result in a tight event arrival function description for head-controlled loops.*

Therefore, the number of flows into the loop body is restricted for head-controlled loops using the following constraint:

$$\forall \ell \in \mathcal{L}_{\mathrm{H}} : \sum_{i \in \mathcal{E}_\ell^{\mathrm{r}}} \sum_{j \in (\mathcal{S}_i \cap \mathcal{M}_\ell)} p_{i,j} \leq n_\ell^H \tag{5.26}$$

The set $\mathcal{L}_{\mathrm{H}}$ contains all head-controlled loops, whereas the set $\mathcal{S}_i$ contains all directly succeeding basic blocks of basic block i. $\mathcal{E}_\ell^{\mathrm{r}}$ contains the *regular* entrance block of the loop ℓ ($|\mathcal{E}_\ell^{\mathrm{r}}| = 1$, as there is only one regular entry per loop). The loops depicted in Figure 5.4(a) and Figure 5.4(b) only have one loop entry (basic block `B1`), which is regular. Yet, the exemplary head-controlled loop structure shown in Figure 5.4(c) has two loop entrances: One *regular* entrance (`B1`) and one *irregular* entrance (`B2`). Irregular loop entrances can stem from, e.g., `goto`-instructions into the body of a loop. Equation (5.26) restricts the sum of all flows into the loop body by its regular entrance to be less than or equal to n_ℓ^H. This upper limit is set as follows:

$$n_\ell^H = \underbrace{\mathrm{B}_\ell^{\mathrm{Up}} \cdot \left(\sum_{i \in \mathcal{E}_\ell} \sum_{j \in (\mathcal{P}_i \setminus \mathcal{M}_\ell)} p_{j,i} + s_\ell \right) - s_\ell}_{\mathrm{I}} - \underbrace{\sum_{i \in \mathcal{E}_\ell^{\mathrm{i}}} \sum_{j \in (\mathcal{P}_i \setminus \mathcal{M}_\ell)} p_{j,i}}_{\mathrm{II}} \tag{5.27}$$

The first part of Equation (5.27) (marked with "I") is similar to the constraint Equation (5.23), yet with minor modifications. For every flow arriving at the loop, the loop body is allowed to be entered $\mathrm{B}_\ell^{\mathrm{Up}}$ times. In case the sub-path starts inside the loop ($s_\ell = 1$), the loop body is allowed to be entered another $(\mathrm{B}_\ell^{\mathrm{Up}} - 1)$ times. The deduction of one iteration in this case stems from the fact, that when the sub-path starts inside the loop, the loop is at least (partially) executed once.

The second term (marked with "II") of Equation (5.27) handles flows arriving from irregular loop entrances. In case the loop is entered using an irregular loop entrance, this arriving flow clearly does not increase the number of times the loop body is entered via a regular entrance. As the maximum execution count of head-controlled loops is controlled by limiting the number of flows into the loop via its regular entrance (cf. Equation (5.26)), the maximum count variable n_ℓ^H is reduced for every flow arriving from an irregular entrance. Irregular tail-controlled loops do not require this additional term, as there the execution count of the back-edges is restricted. It is therefore irrelevant, from where the tail-controlled loop is entered, as the same back-edges are used.

With these additional constraints, the maximum execution count of loop structures is bounded. Additionally, the precision of an event arrival function can be increased if also the lower bound $\mathrm{B}_\ell^{\mathrm{Low}}$ of a loop ℓ is considered inside the model.

Example 5.4. *A program only consisting of the loop structure depicted in Figure 5.4(a) is assumed, whereas the loop has an upper loop bound of* $\mathrm{B}_\ell^{Up} = 5$ *and a lower loop bound of* $\mathrm{B}_\ell^{Low} = 3$*. Furthermore, it is assumed that only basic blocks* `B0` *and* `B4` *generate one event each. When only restricting the maximum number of loop iterations, the sub-path* $\{$`B0`, `B1`, `B2`, `B3`, `B4`$\}$ *would be valid and chosen as the shortest possible sub-path to generate 2 events. While this would be a safe over-approximation (as* η^+ *would never be under-approximated), it would create a pessimistic* η^+*. When considering lower loop bounds, the shortest sub-path covering two events would be* $\{$`B0`, `B1`, `B2`, `B3`, `B1`, `B2`, `B3`, `B1`, `B2`, `B3`, `B4`$\}$.

A new set $\mathcal{X}_\ell$ is introduced, containing all exiting blocks of a loop ℓ. A loop exit block is defined as a basic block belonging to a loop ℓ with a successor which is not part of the loop. Referring to Figure 5.4(a), basic block `B3` is a loop exit, whereas in Figure 5.4(b), basic block `B1` is a loop exit. Additionally, a binary constant T_ℓ is introduced, representing whether a loop ℓ is tail-controlled ($T_\ell = 1$) or not ($T_\ell = 0$). Using the following constraints which are added for every loop, lower loop bounds are considered:

$$\forall \ell \in \mathcal{L} : g_\ell = \sum_{i \in \mathcal{E}_\ell} \sum_{j \in (\mathcal{P}_i \setminus \mathcal{M}_\ell)} p_{j,i} \tag{5.28}$$

$$h_\ell = \sum_{i \in \mathcal{X}_\ell} \sum_{j \in (\mathcal{S}_i \setminus \mathcal{M}_\ell)} p_{i,j} \tag{5.29}$$

$$\sum_{(i,j) \in \mathcal{N}_\ell} p_{i,j} \geq \min(g_\ell, h_\ell) \cdot (\mathrm{B}_\ell^{\mathrm{Low}} - T_\ell) \tag{5.30}$$

The variables g_ℓ and h_ℓ introduced in Equation (5.28) and Equation (5.29) are only present for a better readability of Equation (5.30). g_ℓ denotes all flows arriving at the loop head of loop ℓ, whereas h_ℓ represents all flows exiting the loop. The term $\min(g_\ell, h_\ell)$ denotes the number of times a loop is entered *and* exited. This is relevant, as a sub-path may start or end inside a loop, thus a loop may be entered (resp. exited) on a sub-path, but not exited (resp. entered). For each time a loop is entered and exited, the back-edges of this loop (the set $\mathcal{N}_\ell$ denotes all back-edges of a loop ℓ) have to be executed at least $\mathrm{B}_\ell^{\mathrm{Low}}$ times (or $\mathrm{B}_\ell^{\mathrm{Low}} - 1$ for a tail-controlled loop).

5.3.4 Flow Facts

Flow facts restrict the set of possible execution scenarios [Kir03]. Using flow facts, upper or lower loop bounds can be expressed, as well as maximum recursion depths or in general, relationships between control-flows in a given program. As an illustrative example, Figure 5.5 shows a control-flow graph with the recursive function `fac`. Without further information, determining the WCET is impossible, as the maximum recursion depth of the function `fac` is unknown. If the maximum recursion depth of `fac` is known to be less than or equal to 10, this can be described using the following inequation, a so-called flow fact:[1]

$$1 \cdot p_{\mathtt{B2}} \leq 11 \cdot p_{\mathtt{B0}} \tag{5.31}$$

[1] Note that `fac` is called up to 11 times here, as it is once called from `main`.

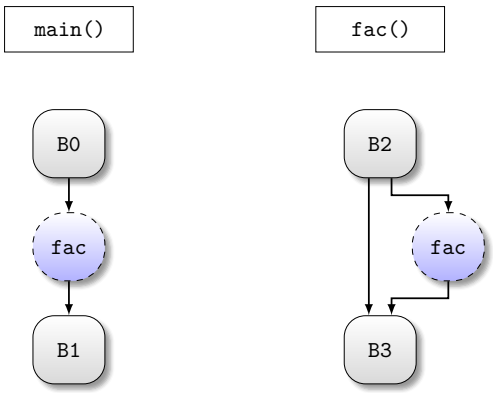


Figure 5.5. – An exemplary control-flow graph with a recursive function.

Where $p_{\texttt{B2}}$ is the total execution count of basic block B2 and $p_{\texttt{B0}}$ the execution count of B0. Similarly, nested triangular loops can be described, resulting in a more precise description than simple loop bounds. In the well-known IPET approach [LM95], these constraints can simply be integrated into the ILP. Yet, as a sub-path may start, e.g., inside a called function, these flow fact constraints have to be modified, as they otherwise may impose faulty restrictions on the sub-path. For example, Equation (5.31) would forbid a sub-path to start at B2 ($p_{\texttt{B0}} = 0 \Rightarrow p_{\texttt{B2}} \leq 0$). Therefore, flow facts require a slight extension in order to be integrated into the presented ILP model of an event arrival function.

It is assumed that a flow fact is represented by a constraint in the following form (which is not directly integrated into the ILP formulation):

$$X \cdot p_i \leq Y \cdot p_j, \tag{5.32}$$

$$X, Y \in \mathbb{N}_0, X \leq Y \tag{5.33}$$

where i and j are basic blocks. For each flow fact constraint, a binary ILP variable v_j is created, representing whether basic block j (right hand-side basic block of Equation (5.32)) is not executed on the sub-path ($v_j = 1$) or it is ($v_j = 0$):

$$v_j = \begin{cases} 0 & \text{if } \sum\limits_{n \in \mathcal{P}_j} p_{n,j} \geq 1, \\ 1 & \text{else.} \end{cases} \tag{5.34}$$

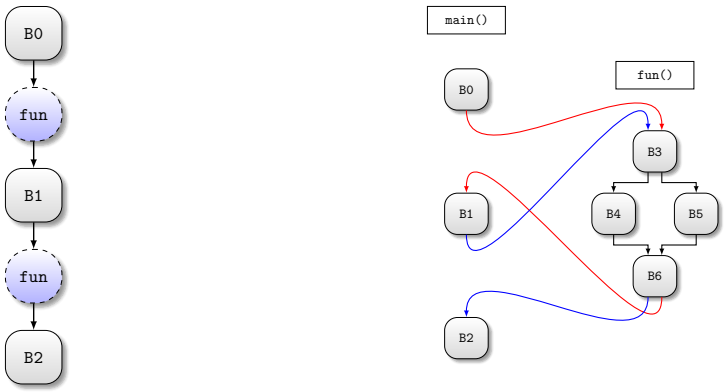
This variable is used to create a pseudo-flow f_j, such that in case the referenced basic block j is never executed on the sub-path, the flow fact can still hold:

$$f_j = \sum_{n \in \mathcal{P}_j} p_{n,j} + v_j \tag{5.35}$$

Subsequently, each flow fact is inserted into the ILP model in a modified way:

$$X \cdot p_i \leq Y \cdot f_j \tag{5.36}$$

Equation (5.36) replaced the flow variable p_j of the right side of Equation (5.32) by the corresponding pseudo-flow f_j. Therefore, no possible paths are excluded, even



(a) Control-flow graph with function calls as stubs. **(b)** Control-flow graph with interprocedural edges.

Figure 5.6. – Exemplary CFG with two calls to function `fun`, once with calls shown as simple stubs (a) and once with the actual control-flow edges (b).

if basic block j is not part of the sub-path. Yet, if it is on the sub-path, the flow fact holds as intended.

5.3.5 Function Calls

Function calls can easily be expressed in the model by simply treating the called function (or the first basic block of the function to be more precise) like a successor to the basic block with the call. Additionally, the basic block succeeding the call is added as successor to all function-exiting blocks of the called function. This is displayed with an example in Figure 5.6. Figure 5.6(a) shows a simple exemplary control-flow graph, where `B0` ends with a call to a function `fun`, whereas basic block `B1` continues after the function `fun` returns. Basic block `B1` again ends with a call to `fun`, whereas basic block `B2` continues with the execution after `fun` returns. Figure 5.6(b) shows the actual control-flow graph of function `fun` and adds control-flow edges to the basic blocks ending with a call and the corresponding return-edges from the called function. The call- and return-edges which belong together (e.g., $\texttt{B0} \rightarrow \texttt{B3}$ and $\texttt{B6} \rightarrow \texttt{B1}$) are marked in the same color. Obviously, a sub-path over call- and return-edges which do not belong together (e.g., $\{\texttt{B0}, \texttt{B3}, \texttt{B4}, \texttt{B6}, \texttt{B2}\}$) should be excluded. The basic IPET [LM95] approach enforces this by forcing the call-edge and return-edge to be executed the same number of times. Yet, as a sub-path may start inside a called function (or further nested function calls), the number of times a call-edge and the corresponding return-edges are executed may differ (e.g., $\{\texttt{B4}, \texttt{B6}, \texttt{B2}\}$ is a valid sub-path). Even more, in case of a recursive function, a call-edge (resp. return-edge) may be executed several times, but the corresponding return-edge (resp. call-edge) may be executed not a single time, yet it would still be valid sub-path.

In case of non-recursive functions (e.g., as shown in Figure 5.6) the following constraints are added to the ILP to enforce valid sub-paths:

$$\forall \gamma \in \mathcal{F} : \forall (i,j) \in \mathcal{C}_\gamma : p_{i,j} \geq \left(\sum_{(m,n) \in \mathcal{R}_{\gamma,(i,j)}} p_{m,n} \right) - s_\gamma \tag{5.37}$$

$$\sum_{(m,n) \in \mathcal{R}_{\gamma,(i,j)}} p_{m,n} \geq p_{i,j} - e_\gamma \tag{5.38}$$

The set $\mathcal{F}$ contains all functions of the current program, whereas $\mathcal{C}_\gamma$ contains all call-edges to a function γ. Referring to Figure 5.6(b), $\mathcal{C}_{\texttt{fun}}$ would be $\{\texttt{B0} \rightarrow \texttt{B3}, \texttt{B1} \rightarrow \texttt{B3}\}$. $\mathcal{R}_{\gamma,(i,j)}$ contains all return-edges of function γ when called via the edge (i,j). Referring to Figure 5.6(b), $\mathcal{R}_{\texttt{fun},(\texttt{B0},\texttt{B3})}$ is $\{\texttt{B6} \rightarrow \texttt{B1}\}$. The binary variable s_γ is set to 1 in case the sub-path starts in the function γ (cf. Equation (5.25)).

Equation (5.37) restricts the calling edge to a function to be executed not less than the number of times the corresponding return edges are executed. In case the start of the sub-path is chosen inside the called function (or in a nested function called), the calling edge may be executed one time less than the return edges. Similarly, Equation (5.38) enforces the corresponding return edges to be executed at least as often as the call edge, unless the sub-path ends inside the called function. These constraints are illustrated using two different sub-paths of the exemplary CFG shown in Figure 5.6(b), one being allowed and one forbidden:

Example 5.5 (Sub-Path $\{\texttt{B4}, \texttt{B6}, \texttt{B2}\}$)**.** *This sub-path starts in the function* `foo()`*, ends in the function* `main()` *and is a valid sub-path. The corresponding constraints limiting the call- and return-edges for this sub-path (cf. Equations* (5.37) *to* (5.38)*) are as follows:*

$$p_{B0,B3} \geq p_{B6,B1} - s_{fun} \Rightarrow 0 \geq 0 - 1 \tag{5.39}$$

$$p_{B6,B1} \geq p_{B0,B3} - e_{fun} \Rightarrow 0 \geq 0 - 0 \tag{5.40}$$

$$p_{B1,B3} \geq p_{B6,B2} - s_{fun} \Rightarrow 0 \geq 1 - 1 \tag{5.41}$$

$$p_{B6,B2} \geq p_{B1,B3} - e_{fun} \Rightarrow 1 \geq 0 - 0 \tag{5.42}$$

As the minimum number of executions of the corresponding call-edge is decreased by one since the sub-path is starting in the called function, the constraints are not contradicting and the sub-path is allowed.

Example 5.6 (Sub-Path $\{\texttt{B0}, \texttt{B3}, \texttt{B4}, \texttt{B6}, \texttt{B2}\}$)**.** *This sub-path starts in the function* `main()` *at basic block* `B0`*, includes one execution of the function* `foo()` *and ends at basic block* `B2` *after exiting function* `foo()`*. It is an invalid sub-path, since the function* `foo()` *is called from basic block* `B0`*, therefore the control flow has to return to* `B1` *after the call of* `foo()`*, not to* `B2`*. The corresponding constraints limiting the call- and return-edges for this sub-path (cf. Equations* (5.37) *to* (5.38)*) are as follows:*

$$p_{B0,B3} \geq p_{B6,B1} - s_{fun} \Rightarrow 1 \geq 0 - 0 \tag{5.43}$$

$$p_{B6,B1} \geq p_{B0,B3} - e_{fun} \Rightarrow 0 \geq 1 - 0 \quad \lightning \tag{5.44}$$

$$p_{B1,B3} \geq p_{B6,B2} - s_{fun} \Rightarrow 0 \geq 1 - 0 \quad \lightning \tag{5.45}$$

$$p_{B6,B2} \geq p_{B1,B3} - e_{fun} \Rightarrow 1 \geq 0 - 0 \tag{5.46}$$

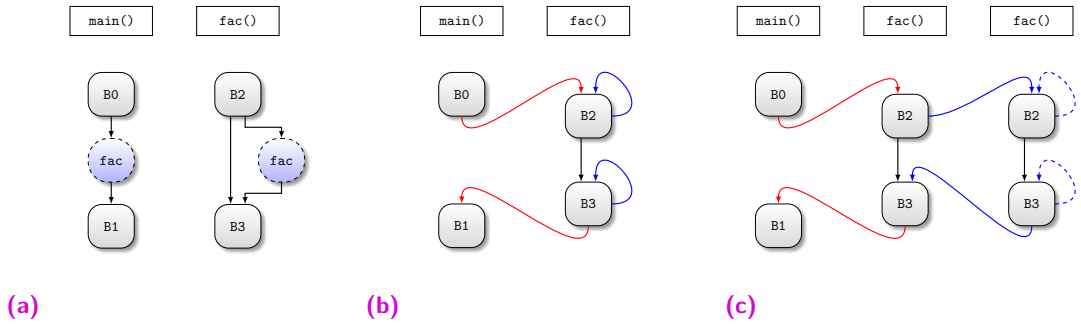


Figure 5.7. – An exemplary CFG with a recursive function. Figure (a) shows function calls only as stubs, whereas Figure (b) includes interprocedural control-flow edges. Figure (c) shows the interprocedural control-flow edges with a single recursive call of function `fac()` virtually unrolled.

Equations (5.44) *and* (5.45) *do not hold, therefore this invalid sub-path cannot be chosen by the ILP solver.*

While these constraints can handle simple function calls, they are not suitable for covering recursive functions. An example is given in Figure 5.7 with `fac()` being a recursive function. Figure 5.7(a) shows the control-flow graphs for both functions `main()` and `fac()` separated, whereas Figure 5.7(b) shows a control-flow graph with interprocedural edges. These interprocedural edges are generated in the same manner as for non-recursive functions, meaning the potential call targets of a call are added as successors of the caller block in the control-flow graph (and all exiting blocks of the potentially called functions as predecessors to the basic block succeeding the calling basic block). Since basic basic block `B2` ends with a conditional call to the function `fac()` itself, `B2` has two successors: `B3` if the call is not executed and `B2` (itself, as it is the first block of the function `fac()`). Thus `B2` has an edge to itself in Figure 5.7(b). The call to `fac()` from `fac()` returns to `B3`, which itself is the exit block of the function `fac()`. Therefore, also `B3` has an edge to itself in Figure 5.7(b). For a better overview of the recursive edges of `fac()`, another CFG with interprocedural edges and a single recursive call of the function `fac()` virtually unrolled is shown in Figure 5.7(c). This virtual unrolling is only done here for a better visualization of the valid control flows when `fac()` called or returned from, this is not applied during the ILP generation. Here, the sub-path $\{\texttt{B3}, \texttt{B3}, \texttt{B3}, \texttt{B1}\}$ is valid, as a sub-path may start inside a nested function call of `fac()`. Yet, this sub-path would violate Equation (5.37), as the call-return-ratio can only be reduced by 1 in case the sub-path starts inside a function. Therefore, the following constraints are added to the ILP in case of recursive functions:

$$\forall \gamma \in \mathcal{F} : \forall (i,j) \in \mathcal{C}_\gamma : p_{i,j} \geq \min \left(\sum_{(m,n) \in \mathcal{R}_{\gamma,(i,j)}} p_{m,n}, \sum_{(x,y) \in \mathcal{C}_\gamma} p_{x,y} \right) - s_\gamma \tag{5.47}$$

$$\sum_{(m,n) \in \mathcal{R}_{\gamma,(i,j)}} p_{m,n} \geq \min \left(p_{i,j}, \sum_{(x,y) \in \mathcal{R}_\gamma} p_{x,y} \right) - e_\gamma \tag{5.48}$$

The set $\mathcal{R}_\gamma$ contains all return-edges from the function γ. Therefore, $\mathcal{R}_{\gamma,(i,j)}$ is a subset of $\mathcal{R}_\gamma$ ($\mathcal{R}_{\gamma,(i,j)} \subseteq \mathcal{R}_\gamma$). Referring to Figure 5.7, $\mathcal{R}_\gamma$ is $\{\texttt{B3} \rightarrow \texttt{B1}, \texttt{B3} \rightarrow \texttt{B3}\}$, whereas $\mathcal{R}_{\texttt{fac},(\texttt{B0},\texttt{B2})}$ is $\{\texttt{B3} \rightarrow \texttt{B1}\}$.

Equation (5.47) sets a lower bound on every calling edge. The first term inside the min-term and the subtractive s_γ are identical to the non-recursive case. As the difference between the number of executions of a call-edge and the sum of the corresponding return-edges can be greater than 1 for recursive functions, the min-term evaluates the minimum number of executed call-edges for a valid control-flow. The general idea is that a return-edge can be executed multiple times without any call-edge on the sub-path (e.g., when returning from a recursive function multiple times), yet *if* there is a call-edge to the function on the sub-path, it has to be the corresponding one to the return-edge on the sub-path. This special behavior of recursive functions is handled by the second term inside the min-term of Equation (5.47). Referring to the CFG in Figure 5.7, the sub-path $\{\texttt{B2}, \texttt{B3}, \texttt{B3}, \texttt{B1}\}$ would be valid, as it would occur during a call chain of `main()`→`fac()`→`fac()`. As long as there is no call to `fac()` on the sub-path, the min-term will always evaluate to 0, enabling the previously mentioned sub-path. Yet, a sub-path $\{\texttt{B0}, \texttt{B2}, \texttt{B3}, \texttt{B3}, \texttt{B1}\}$ (note the addition of `B0` at the beginning) would be feasible according to the simple edges of the CFG (cf. Figure 5.7(b)), but not actually valid, as there is a recursive call to `fac()` (`B2`→`B2`) missing on the sub-path. Since there is a call to `fac()` on the sub-path, as well as a return from a recursive call (`B3`→`B3`), the min-term in Equation (5.47) will evaluate to 1 for the recursive call to `fac()`, setting a lower bound of 1 to the calling edge (`B2`→`B2`). Essentially, the second term inside the min-term in Equation (5.47) only comes into play to ensure valid combinations of call- and return-chains of recursive functions. It makes certain, that multiple returns of a recursive call may exist on the sub-path. Yet, as soon as there is a call to this function on the sub-path, it has to be part of a valid call- and return-chain, which is done by comparing the overall number of calls to this function and the calling-edge-specific returns.

While Equation (5.47) enforces a minimum on the number of times a call-edge is executed, Equation (5.48) sets a lower limit for the return-edges. The idea is in analogy to the call-edges. The first term inside the min-term and the subtractive e_γ is identical to the non-recursive case. In case of a recursive function, a call-edge (i, j) can be executed multiple times on a sub-path without any return-edge executed. Yet, *if* a return-edge is part of the sub-path, it should be a fitting one. This is then enforced by setting a lower bound on the sum over all corresponding return-edges. Referring to Figure 5.7, a sub-path like $\{\texttt{B2}, \texttt{B2}, \texttt{B2}, \texttt{B3}\}$ with multiple calls to `fac()` is a valid sub-path. Here, the second term in the min-term in Equation (5.48) would evaluate to 0, as there are no return-edges of `fac()` on the sub-path. This way, Equation (5.48) does not impose a minimum execution count on any of the function's return-edges. If a return-edge of `fac()` becomes part of the sub-path (e.g., $\{\texttt{B2}, \texttt{B2}, \texttt{B2}, \texttt{B3}, \texttt{B1}\}$), the second term inside the min-term in Equation (5.48) is no longer 0 and it is dependent on the execution count of the corresponding call-edge, to what the min-term evaluates. When looking at the recursive call-edge (`B2`→`B2`), the corresponding return-edge (`B3`→`B3`) has to be executed at least as many times as the recursive call-edge, or the overall number of return-edges of the function, whichever is lower (assuming the sub-path does not end inside `fac()`). If the overall number of executed return-edges of `fac()` is lower or equal to the execution count of the recursive call-edge, all of

those executed return-edges have to be corresponding to the recursive call-edge. Otherwise, this would represent an invalid call-chain, as at least one level of recursion was skipped in terms of executed returns.

These constraints are illustrated using two different sub-paths of the exemplary CFG shown in Figure 5.7, one being allowed and one forbidden:

Example 5.7 (Sub-Path $\{\texttt{B3}, \texttt{B3}, \texttt{B3}, \texttt{B1}\}$)**.** *This valid sub-path starts in an inner recursive function call of* `fac()`*, includes two returns from and into* `fac()` *and then ends after returning to the function* `main()`*. The corresponding constraints handling the calls and returns (cf. Equations* (5.47) *and* (5.48)*) are as follows for the example:*

$$p_{B0,B2} \geq \min\left(p_{B3,B1}, p_{B2,B2} + p_{B0,B2}\right) - s_{fac} \Rightarrow 0 \geq \min(1,0) - 1 \tag{5.49}$$
$$p_{B3,B1} \geq \min\left(p_{B0,B2}, p_{B3,B3} + p_{B3,B1}\right) - e_{fac} \Rightarrow 1 \geq \min(0,3) - 0 \tag{5.50}$$
$$p_{B2,B2} \geq \min\left(p_{B3,B3}, p_{B2,B2} + p_{B0,B2}\right) - s_{fac} \Rightarrow 0 \geq \min(2,0) - 1 \tag{5.51}$$
$$p_{B3,B3} \geq \min\left(p_{B2,B2}, p_{B3,B3} + p_{B3,B1}\right) - e_{fac} \Rightarrow 2 \geq \min(0,3) - 0 \tag{5.52}$$

As all constraints are valid, the ILP solver may choose this sub-path.

Example 5.8 (Sub-Path $\{\texttt{B0}, \texttt{B2}, \texttt{B2}, \texttt{B3}, \texttt{B1}\}$)**.** *This sub-path starts in* `main()`*, then enters the function* `fac()` *twice, and returns back to* `main()`*. It is an invalid path, as one return from* `fac()` *to* `fac()` *is skipped, yet potentially possible when only considering the edges of the CFG (see Figure 5.7(b)). The corresponding constraints handling the calls and returns (cf. Equations* (5.47) *and* (5.48)*) are as follows for this example:*

$$p_{B0,B2} \geq \min\left(p_{B3,B1}, p_{B0,B2} + p_{B2,B2}\right) - s_{fac} \Rightarrow 1 \geq \min(1,2) - 0 \tag{5.53}$$
$$p_{B3,B1} \geq \min\left(p_{B0,B2}, p_{B3,B1} + p_{B3,B3}\right) - e_{fac} \Rightarrow 1 \geq \min(1,1) - 0 \tag{5.54}$$
$$p_{B2,B2} \geq \min\left(p_{B3,B3}, p_{B2,B2} + p_{B0,B2}\right) - s_{fac} \Rightarrow 1 \geq \min(0,2) - 0 \tag{5.55}$$
$$p_{B3,B3} \geq \min\left(p_{B2,B2}, p_{B3,B3} + p_{B3,B1}\right) - e_{fac} \Rightarrow 0 \geq \min(1,1) - 0 \quad \lightning \tag{5.56}$$

Since this sub-path violates Equation (5.56)*, the ILP solver cannot choose this invalid sub-path.*

Note, that these constraints only handle the correct execution count ratio of call- and return-edges. Actual limits on, e.g., maximum recursion depths, are handled by flow facts as discussed in the previous section Section 5.3.4.

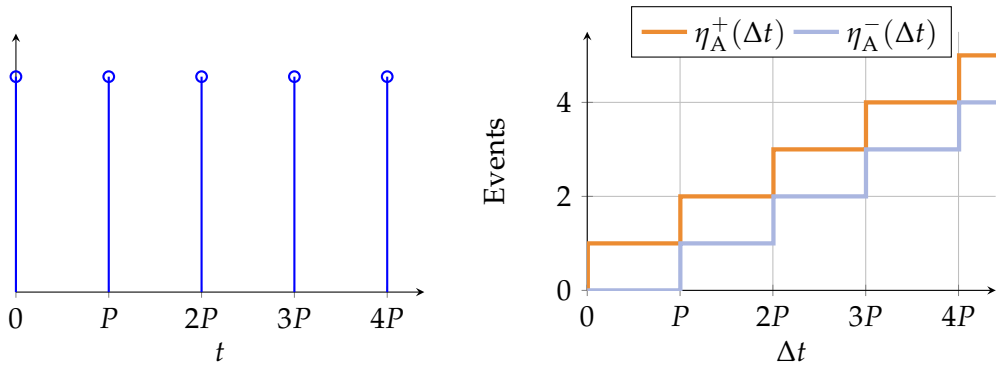
5.3.6 Arbitrarily Activated Tasks

In most cases, a task is not only executed once in the whole lifetime of the system, but repeatedly in a somewhat periodic fashion. Before explaining how the ILP model can be extended to support arbitrarily activated tasks, a brief introduction into describing task activation patterns is given.

Task Models

In order to describe the periodic behavior of a task, numerous so-called task models have been established in the literature [Ric04]:

- Strictly periodic tasks. Each task is described by its period, after which its execution is repeated.



(a) A periodic task activiation with a period of P in absolute time.

(b) The corresponding event arrival functions $\eta_A^+(\Delta t)$ and $\eta_A^-(\Delta t)$.

Figure 5.8. – A strictly periodic task activation pattern (a) and its corresponding event arrival functions (b).

- Periodic tasks with jitter. Here, a task has a nominal period, yet each activation may be delayed by a certain time relative to its reference activation period. The upper bound of this delay is the so-called *jitter*.
- Periodic tasks with jitter and a defined minimum interarrival time. Additionally to a periodic task model with jitter, this task model defines a minimum interarrival time between two executions of a task.

All these task models can also be described by corresponding event arrival functions. In the following, the upper event arrival function used to describe the *activation pattern* of a task will be denoted as $\eta_A^+(\Delta t)$. Therefore, the event arrival function $\eta^+(\Delta t)$ to be *derived* from within a task should not be confused with the event arrival function $\eta_A^+(\Delta t)$ describing the activation pattern of the same task. The same applies respectively for the lower event arrival functions.

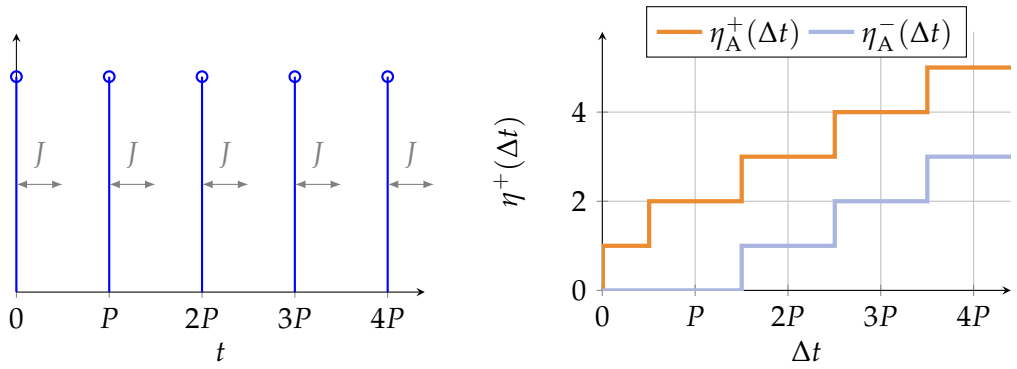
A task's activation pattern cannot only be described by the maximum number of activations per time interval Δt, but also using the minimum time interval between n task activations. This pseudo-inverse function of the upper event arrival function $\eta_A^+(\Delta t)$ is named the lower distance function $\delta_A^-(n)$. An upper event arrival function $\eta_A^+(\Delta t)$ can be easily transformed into its equivalent lower distance function $\delta_A^-(n)$ and vice versa [DAE12].

Example 5.9 (Event arrival functions of a strictly periodic task)**.** *Figure 5.8(a) shows the activation pattern of a strictly periodically executed task. The task is activated every P time units. Beside, Figure 5.8(b) shows the corresponding upper (resp. lower) event arrival function $\eta_A^+(\Delta t)$ (resp. $\eta_A^-(\Delta t)$) of this task activation pattern. In general, the upper event arrival function of a strictly periodic task is described by the equation:*

$$\forall \Delta t \geq 0 : \eta_A^+(\Delta t) = \left\lceil \frac{\Delta t}{P} \right\rceil \tag{5.57}$$

And similarly for the lower event arrival function:

$$\forall \Delta t \geq 0 : \eta_A^-(\Delta t) = \left\lfloor \frac{\Delta t}{P} \right\rfloor \tag{5.58}$$



(a) A periodic task activiation with a period of P and a jitter of J in absolute time.

(b) The corresponding event arrival functions $\eta_A^+(\Delta t)$ and $\eta_A^-(\Delta t)$.

Figure 5.9. – A periodic task with jitter activation pattern (a) and its corresponding event arrival functions (b).

The corresponding lower and upper distance functions are as follows:

$$\delta_A^-(n) = \delta_A^+(n) = \max((n-1)\cdot P, 0) \tag{5.59}$$

Example 5.10 (Event arrival function of a periodic task with jitter)**.** *Figure 5.9(a) shows the activation pattern of a periodic task with jitter. The task has an associated period of P, whereas a jitter with a maximum length of J adds a certain uncertainty on the exact activation of a task. An activation of a task can be delayed by up to J time units after the "actual" strictly periodical activation defined by the task's period. The corresponding event arrival functions are shown beside in Figure 5.9(b). Note that the upper event arrival function $\eta_A^+(\Delta t)$ is shifted by J time units towards the y-axis, whereas the lower event arrival function $\eta_A^-(\Delta t)$ is shifted by J time units away from the y-axis. In an extreme case, a task activation experiences the full delay of J time units, whereas the next activation has no delay. This shrinks the minimum time interval in which 2 task activations may happen by J time units. Similarly, if a task activation experiences no delay, whereas the subsequent task activation has a delay of J time units, the maximum possible time interval in which 0 task activations happen increases by J time units.*

The upper event arrival function is described by the following equations:

$$\Delta t = 0 : \eta_A^+(\Delta t) = 0 \tag{5.60}$$

$$\forall \Delta t > 0 : \eta_A^+(\Delta t) = \left\lceil \frac{\Delta t + J}{P} \right\rceil \tag{5.61}$$

And similarly for the lower event arrival function:

$$\forall \Delta t \geq 0 : \eta_A^-(\Delta t) = \max\left(\left\lfloor \frac{\Delta t - J}{P} \right\rfloor, 0 \right) \tag{5.62}$$

The lower distance function $\delta_A^-(n)$ can be described as follows:

$$\delta_A^-(n) = \max((n-1)\cdot P - J, 0) \tag{5.63}$$

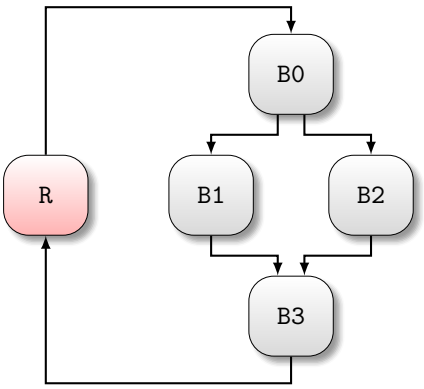


Figure 5.10. – Adapted control-flow graph with periodic edges.

Similarly, the upper distance function $\delta_A^+(n)$*:*

$$\delta_A^+(n) = \max((n-1) \cdot P + J, 0) \tag{5.64}$$

Beside these well-established task models, basically any arbitrary activation pattern can be described by an event arrival function (or its corresponding distance function).

ILP Model Additions for Upper Event Arrival Functions

This section introduces the actual additions to the ILP model of a task's upper event arrival function $\eta^+(\Delta t)$ necessary to integrate an arbitrary task activation pattern. In case of a periodic task, the control-flow graph is virtually adapted to reflect the periodic execution. The term "virtually adapted" means in this case that the modifications are only done in the analysis' control-flow graph, whereas the actual program remains untouched. Figure 5.10 shows the adapted control-flow graph of a periodic task. The gray blocks represent the actual basic blocks of the program, whereas the red block with the label `R` is an additional "virtual" block, representing the time spent in between executing the task. In the following, this block will be referred to as the *idle block*. Additional edges from every exiting block (here basic block `B3`) of the program to the idle block `R` are generated. Furthermore, an edge from the idle block `R` to the program's entrypoint (here basic block `B0`) is created as well. Following the same methods as for regular blocks, a point law constraint as shown in Equation (5.4) is created for the block `R` and corresponding variables for the edges towards/from block `R` are inserted. Yet, the handling of the special block `R` differs in two points:

1. No starting edge indicator variable $s_{\mathtt{R},i}$ is required for the edge from block `R` to the program's entrypoint i. A sub-path can already start at the program's first block by using the edge from the virtual source $\ominus$ to the entrypoint.

2. Block `R` generates 0 events and has no associated execution time. Instead of having a fixed execution time, block `R` can contribute an arbitrary amount of time to the sub-path:

$$z_{\mathtt{R}}^+ \geq 0 \tag{5.65}$$
$$a_{\mathtt{R}}^+ = 0 \tag{5.66}$$

Additionally, the minimum time required for a complete task execution which generates the maximum number of events is derived. This constant is denoted as C_{E}^{-}. It can be determined by using the existing ILP formulation and temporarily modifying it as described in the following steps:

1. Disable the task's virtual periodic edges (from exit blocks to $\mathtt{R}$ and from $\mathtt{R}$ to the task's entry block) by forcing the number of flows into the idle block $\mathtt{R}$ to be zero:

$$\sum_{i \in \mathcal{P}_{\mathtt{R}}} p_{i,\mathtt{R}} = 0 \tag{5.67}$$

2. Force a complete path through the program by setting the sub-path to start at the entrypoint j and end at an ending block:

$$s_{\ominus,j} = 1 \tag{5.68}$$

$$\sum_{n \in \mathcal{T}} e_n = 1 \tag{5.69}$$

3. Remove the subtractive term from the accumulated timing constraint (see Equation (5.10)) from the entrypoint basic block and all sink basic blocks. This is done, as C_{E}^{-} represents the minimum time for one *complete* task execution. E.g., the accumulated timing constraint for a sink basic block i would be simplified to the following (note the missing subtractive term on the right-hand side):

$$z_i^{+} \geq C_i^{-} \cdot \sum_{j \in \mathcal{P}_i} p_{j,i} \tag{5.70}$$

4. Set the time-interval constant Δt from Equation (5.12) to a sufficiently large constant M to cover a complete path (e.g., the WCET of the program):

$$M \geq \sum_{i \in \mathcal{B}} z_i^{+} \tag{5.71}$$

5. Solve the ILP for maximizing the number of events by using the following objective function, whereas the returned objective value is the maximum number of events generated A_{E}^{+} during one complete task execution:

$$\max : a_{\mathrm{Total}}^{+} \tag{5.72}$$

6. Finally, derive the *minimum* time required to generate this maximum number of events A_{E}^{+} while performing one complete task execution. For this purpose, the number of accumulated events is fixed to A_{E}^{+} and the ILP's objective is replaced to minimize the accumulated time to execute the complete path through the program:

$$a_{\mathrm{Total}}^{+} = A_{\mathrm{E}}^{+} \tag{5.73}$$

$$\min : \sum_{i \in \mathcal{B}} z_i^{+} \tag{5.74}$$

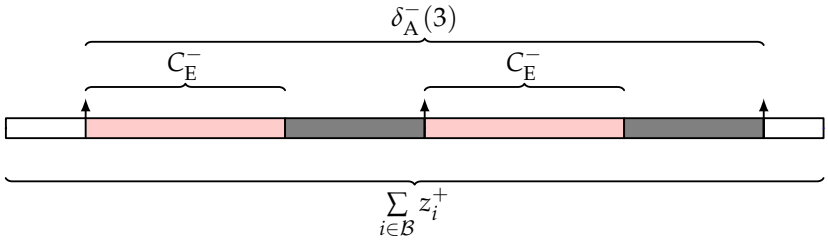


Figure 5.11. – Timing diagram of an exemplary sub-path spanning over 3 task activations.

7. The ILP is solved and the returned objective value is the minimum time for a complete task execution C_{E}^{-} while generating the maximum number of events.

The activation pattern of the task is integrated into the ILP model by enforcing the minimal number of cycles between task activations. This is done using the following constraint, whereas basic block j is the task's entrypoint in the following:

$$\max\left(\sum_{i\in\mathcal{P}_j} p_{i,j} - 1, 0\right) \cdot C_{\mathrm{E}}^{-} + z_{\mathrm{R}}^{+} \geq \delta_{\mathrm{A}}^{-}\left(\sum_{i\in\mathcal{P}_j} p_{i,j}\right) \tag{5.75}$$

The right-hand side of Equation (5.75) results in the minimum time interval required for the number of task activations. As basic block j is assumed to be the task's entrypoint, the number of its executions $\sum_{i\in\mathcal{P}_j} p_{i,j}$ along the sub-path corresponds to the number of task activations. For the sake of brevity, the number of task activations $\sum_{i\in\mathcal{P}_j} p_{i,j}$ along the sub-path is simply denoted as n in the following. If the sub-path spans across n task activations, the sub-path has to include $(n-1)$ complete task executions. A complete task execution is defined as a path starting at the entrypoint of the task and ending at any of the program's ending blocks. This time required for all $(n-1)$ complete task executions is represented by the $\max(\dots) \cdot C_{\mathrm{E}}^{-}$ term of the left-hand side of Equation (5.75).

The basic idea of Equation (5.75) is that one complete task execution, while maximizing the number of events in a limited time interval, will take at least C_{E}^{-} cycles, as this is the minimum time to execute the complete program while generating the maximum number of events. Taking any other path for a complete task execution can only result in an equal or lower number of events, as the remaining time for the rest of the sub-path is lower. Equation (5.75) then enforces the minimal time between task executions by setting a lower bound on the time z_{R}^{+} spent in the idle block R.

Figure 5.11 illustrates Equation (5.75) and is discussed in the following example.

Example 5.11. *In Figure 5.11, the timing diagram of a sub-path spanning across 3 task activations is depicted. The execution of the whole sub-path is depicted as the entire surrounding rectangle, resulting in an accumulated execution time of $\sum_{i\in\mathcal{B}} z_i^{+}$. The upward pointing arrows denote a task activation. The areas filled with a slight red shade depict a complete task execution, whereas the areas filled with gray depict the idle time between task executions. As can be seen in the diagram, the total amount of idle time (i.e., the accumulated time z_R^{+} of the idle block R) has to be at least*

$$z_R^{+} \geq \delta_A^{-}(3) - 2 \cdot C_{\mathrm{E}}^{-}. \tag{5.76}$$

This corresponds to Equation (5.75).

The periodicity constraint Equation (5.75) can be used for any arbitrary task activation pattern, as long as its lower distance function $\delta_{\mathrm{A}}^{-}(n)$ can be described using linear terms (which is the case for all common task activation models). This only requires to express the lower distance function $\delta_{\mathrm{A}}^{-}(n)$ (where n is the number of task activations, hence equal to $\sum_{i \in \mathcal{P}_j} p_{i,j}$ here) of the task's activation pattern inside the ILP model. In the following, this is exemplary shown for a periodic task with jitter.

Example 5.12 (Periodicity constraint for a periodic task with jitter)**.** *As shown in Equation* (5.63)*, the lower distance function $\delta_A^{-}(n)$ for a periodic task with a period of P and a jitter of J is described by the following equation:*

$$\delta_A^{-}(n) = \max((n-1) \cdot P - J, 0) \tag{5.77}$$

As the left-hand side of Equation (5.75) *is always greater than or equal to 0, the* max*-term of Equation* (5.77) *can be dropped. Hence, Equation* (5.75) *would become the following for a periodic task with jitter:*

$$\max\left(\sum_{i \in \mathcal{P}_j} p_{i,j} - 1, 0\right) \cdot C_{\mathrm{E}}^{-} + z_R^{+} \geq \left(\sum_{j \in \mathcal{P}_i} p_{j,i} - 1\right) \cdot P - J \tag{5.78}$$

Equation (5.78) *ensures that if the program's first basic block is executed n times, the time between the first and the nth execution is at least $(n-1) \cdot P - J$ cycles. To reach this minimum time, the idle block* `R` *is used to increase execution time along the sub-path.*

Even for more complex activation patterns (e.g., a task activation pattern with multiple fundamental periods and bursts), the lower distance function $\delta_{\mathrm{A}}^{-}(n)$ can be described using linear terms, hence directly be implemented in the ILP model as additional constraints. For the rare case of a non-linear lower distance function $\delta_{\mathrm{A}}^{-}(n)$, it can be either safely approximated by a linear function or implemented as a quasi-lookup-table inside the ILP.

As the execution time z_{R}^{+} contributed by the idle block `R` is not bound to any incoming or outgoing flows (in contrast to regular basic blocks), theoretically an invalid sub-path (according to the control-flow graph) with additional idle time in between arbitrary basic blocks could be chosen. This is due to z_{R}^{+} only being constrained by a lower bound of 0 (cf. Equation (5.65)) and being part of the overall accumulated execution time of the sub-path. Yet, as the idle block `R` can only *increase* the sub-path's accumulated execution time and does not generate any events, the maximum number of events per given time interval length remains the same.

Overall, the steps for creating the ILP model for an upper event arrival function $\eta^{+}(\Delta t)$ of a periodic task can be summarized as follows:

1. Create the base ILP model for non-periodic tasks as described in Sections 5.3.1 and 5.3.3 to 5.3.5.

2. Adapt the CFG of the program with virtual periodic edges and an idle block `R`. Add the corresponding constraints for these edges and the new block to the ILP model.

3. Determine the maximum number of events during one complete task execution A_{E}^{+} by extending the ILP formulation using Equations (5.67) to (5.72) and solving it.

4. Derive the minimum time required for a complete task execution while generating A_{E}^{+} events, denoted as C_{E}^{-}. This is done by replacing the ILP's objective function by Equation (5.74), adding Equation (5.73) to the ILP and solving the ILP again.

5. Remove the constraints from Equations (5.67) to (5.73) and objective function from Equation (5.74) which were added to derive C_{E}^{-} from the ILP.

6. Using C_{E}^{-} and a linear description of the task's lower distance function $\delta_{\mathrm{A}}^{-}(n)$, add the periodicity constraint from Equation (5.75) to the ILP.

ILP Model Additions for Lower Event Arrival Functions

For deriving a lower event arrival function $\eta^{-}(\Delta t)$ of a periodic task, certain additions and modifications to the base ILP model are required as well. Since ILP's objective function of an upper event arrival function $\eta^{+}(\Delta t)$ requires *maximization* (cf. Equation (5.13)), whereas the ILP's objective of a lower event arrival function $\eta^{-}(\Delta t)$ requires *minimization* (cf. Equation (5.21)), the additions presented in the previous section can not be directly applied to the ILP model of a lower event arrival function.

In contrast to the ILP modeling of the lower event arrival function $\eta^{-}(\Delta t)$ of a non-periodic task, the constraint setting a minimum length of the sub-path (see Equation (5.20)) does not include the additive M-term for periodic tasks:

$$\Delta t \leq \sum_{i \in \mathcal{B}} z_i^{-} \tag{5.79}$$

This "enlarging" term is omitted, as the periodic back-edges circumvent an infeasible model for larger sub-paths.

As a prerequisite, the *maximum* time required for one complete task execution C_{E}^{+} while generating the *minimum* number of events is derived. This is done using the following steps:

1. Disable the task's virtual back-edges by forcing the number of flows into the idle block `R` to be zero:

$$\sum_{i \in \mathcal{P}_{\mathtt{R}}} p_{i,\mathtt{R}} = 0 \tag{5.80}$$

2. Force a complete path through the program by setting the sub-path to start at the entrypoint j and end at an ending block:

$$s_{\ominus,j} = 1 \tag{5.81}$$

$$\sum_{n \in \mathcal{T}} e_n = 1 \tag{5.82}$$

3. Set the binary variables to reduce timing and events at the start and end of the sub-path to 0, such that a complete task execution is modeled:

$$u_s = 0 \tag{5.83}$$
$$u_e = 0 \tag{5.84}$$

4. Set the time-interval constant Δt from Equation (5.79) to 0, such that any path can be chosen:

$$0 \leq \sum_{i \in \mathcal{B}} z_i^- \tag{5.85}$$

5. Solve the ILP for minimizing the number of events, whereas the returned objective value is the minimum number A_{E}^- of events generated during one complete task execution:

$$\min : a_{\mathrm{Total}}^- \tag{5.86}$$

6. Finally, derive the *maximum* time required to generate this minimal number of events A_{E}^- while performing one complete task execution. For this purpose, the number of accumulated events is fixed to A_{E}^- and the ILP's objective is replaced to maximize the accumulated time to execute the sub-path:

$$a_{\mathrm{Total}}^- = A_{\mathrm{E}}^- \tag{5.87}$$
$$\max : \sum_{i \in \mathcal{B}} z_i^- \tag{5.88}$$

7. The ILP is solved and the returned objective value is the maximum time for a complete task execution C_{E}^+ while generating the minimal number of events.

The periodicity of the task is represented using the following constraint which restricts the maximum number of cycles $z_{\mathtt{R}}^-$ spent in the idle block $\mathtt{R}$:

$$z_{\mathtt{R}}^- \leq \max\left(\left(\delta_{\mathrm{A}}^+\left(1 + \sum_{i \in \mathcal{P}_{\mathtt{R}}} p_{i,\mathtt{R}}\right) - C_{\mathrm{E}}^+ \cdot \max\left(\sum_{i \in \mathcal{P}_{\mathtt{R}}} p_{i,\mathtt{R}} - 1, 0\right)\right), \delta_{\mathrm{A}}^+(2)\right) \tag{5.89}$$
$$z_{\mathtt{R}}^- \leq M \cdot \sum_{i \in \mathcal{P}_{\mathtt{R}}} p_{i,\mathtt{R}} \tag{5.90}$$

Equation (5.89) sets an upper limit on the number of cycles spent in the idle block $\mathtt{R}$. The overall idea is as follows: When the idle block $\mathtt{R}$ is once on the sub-path, $z_{\mathtt{R}}^-$ represents the idle time between two task activations. If the idle block $\mathtt{R}$ is twice on the sub-path, $z_{\mathtt{R}}^-$ represents the total idle time between three task activations. More general, if the idle block $\mathtt{R}$ is n times on the sub-path, $z_{\mathtt{R}}^-$ represents the total idle time between $n+1$ task activations. Here, n equals the total number of executions of the idle block $\mathtt{R}$, namely $\sum_{i \in \mathcal{P}_{\mathtt{R}}} p_{i,\mathtt{R}}$. The term $\delta_{\mathrm{A}}^+(1 + \dots)$ in Equation (5.89) describes the overall maximum time between those $n+1$ task activations. Since $z_{\mathtt{R}}^-$ only represents the *idle* time between those task activations, the actual execution time of the sub-path needs to be subtracted. This is where the subtractive term $C_{\mathrm{E}}^+ \cdot \max(\dots)$ of

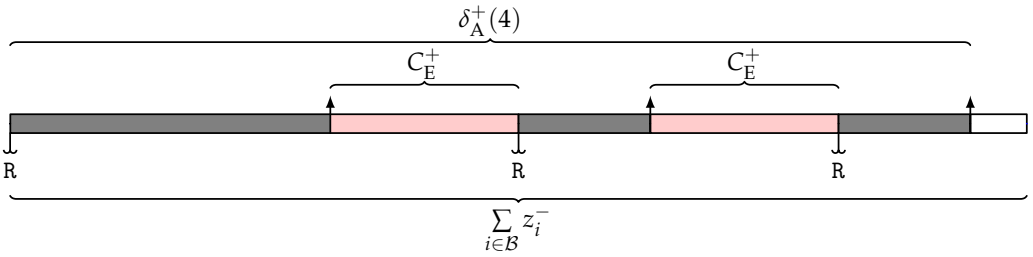


Figure 5.12. – Timing diagram of an exemplary sub-path spanning over 3 task activations.

Equation (5.89) comes into play. If the idle block R is reached two times, a complete task execution is part of the sub-path. When the idle block R is reached three times, two complete task executions are part of the sub-path. More general, if the idle block R is n times part of a sub-path, this sub-path spans over $n-1$ complete task executions. The maximum required execution time $C_E^+ \cdot \max(\dots)$ for these complete task executions is subtracted to derive an upper bound on the total idle time. The 0 of the $\max(\dots,0)$-term avoids *adding* C_E^+ in case the idle block R is not part of the sub-path.

As the sub-path can start at an arbitrary time between two task activations, the outer $\max(\dots,\delta_A^+(2))$-term will set the allowed idle time to at least $\delta_A^+(2)$. Equation (5.90) ensures that idle cycles can only be used to increase the execution time of the sub-path when the idle block R is part of the sub-path. Otherwise, z_R^- could always be set to at least $\delta_A^+(2)$ (and hence contribute to the overall accumulated time of the sub-path) without the sub-path actually required to contain the idle block R.

Figure 5.12 illustrates Equation (5.89) and is discussed in the following example.

Example 5.13. *Figure 5.12 depicts an exemplary sub-path spanning over 3 task activations. The upward pointing arrows denote a task activation. Areas filled with a slight red shade depict a complete task execution, whereas the areas filled with gray depict the idle time between task executions. The ⇂-symbol denotes when the control-flow arrives at the idle block R. The execution of the whole sub-path is depicted as the whole rectangle, resulting in an accumulated execution time of* $\sum_{i\in\mathcal{B}} z_i^-$*. The sub-path starts with an idle period between task executions with the starting point at the idle block R. After each complete task execution, the control-flow reaches the idle block R again, increasing the maximum number of idle cycles* z_R^- *on the sub-path. As can be seen in the diagram, the maximum total number of cycles spent in the idle block in this case is*

$$z_R^- \leq \delta_A^+(4) - 2 \cdot C_E^+. \tag{5.91}$$

This corresponds to Equation (5.89).

Using the periodicity constraint in Equation (5.89), any arbitrary task activation pattern can be included in the ILP model of a lower event arrival function $\eta^-(\Delta t)$ by describing its upper distance function $\delta_A^+(n)$ using linear terms. The following example shows this for a periodic task activation with jitter.

Example 5.14 (Periodicity constraint for a periodic task with jitter for a lower event arrival function)**.** *As shown in Equation* (5.64)*, the upper distance function* $\delta_A^+(n)$ *for a periodic task with a period of* P *and a jitter of* J *is described by the following equation:*

$$\delta_A^+(n) = \max((n-1) \cdot P + J, 0) \tag{5.92}$$

As the right-hand side of Equation (5.89) *is always greater than or equal to 0, the* max*-term of Equation* (5.92) *can be dropped. Hence, Equation* (5.89) *would become the following for a periodic task with jitter:*

$$z_R^- \leq \max\left(\left(\sum_{i \in \mathcal{P}_R} p_{i,R}\right) \cdot P + J - C_{\mathrm{E}}^+ \cdot \max\left(\sum_{i \in \mathcal{P}_R} p_{i,R} - 1, 0\right), P + J\right) \tag{5.93}$$

Overall, the steps for creating the ILP model for a lower event arrival function $\eta^-(\Delta t)$ of a periodic task can be summarized as follows:

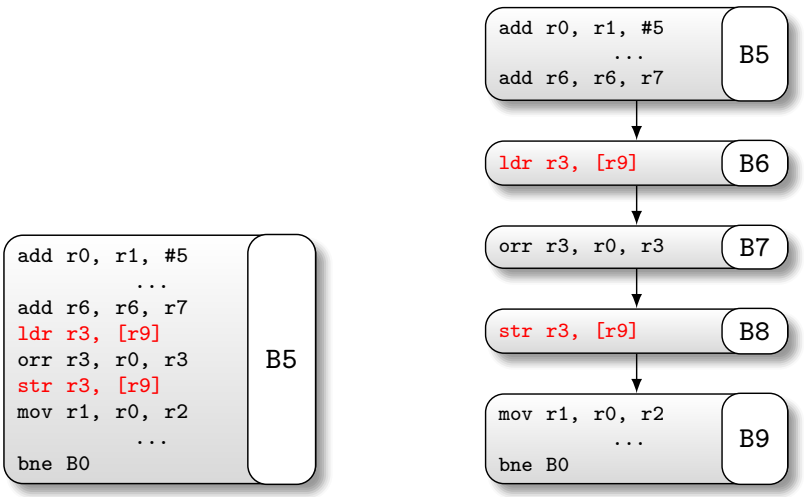
1. Create the base ILP model for non-periodic tasks as described in Sections 5.3.2 to 5.3.5.
2. Adapt the CFG of the program with virtual periodic edges and an idle block R. Add the corresponding constraints for these edges and the new block to the ILP model.
3. Determine the minimum number of events during one complete task execution A_{E}^- by extending the ILP formulation using Equations (5.80) to (5.86) and solving it.
4. Derive the maximum time required for a complete task execution while generating A_{E}^- events, denoted as C_{E}^+. This is done by replacing the ILP's objective function by Equation (5.88), adding Equation (5.87) to the ILP and solving the ILP again.
5. Remove the constraints from Equations (5.80) to (5.87) and objective function from Equation (5.88) which were added to derive C_{E}^+ from the ILP.
6. Using C_{E}^+ and a linear description of the task's upper distance function $\delta_{\mathrm{A}}^+(n)$, add the periodicity constraints from Equations (5.89) and (5.90) to the ILP.

5.3.7 Further Refinements

The model can be further refined to increase the precision of the event arrival function. In the following, additional methods for a tighter model are discussed. While the base ILP model already describes a safe event arrival function, these refinements may increase the tightness of it significantly.

Sub Basic Block Splitting

The granularity of the presented ILP model is on a basic block level. The maximum number of generated events per single execution of basic block i is given by A_i^+. Hence, as soon as basic block i becomes part of the sub-path, the total number of A_i^+



(a) Before sub basic block splitting.

(b) With sub basic block splitting applied.

Figure 5.13. – An exemplary BB containing two event-triggering instructions.

events is added to the accumulated number a^{+}_{Total} of events along the sub-path (as the actual distribution of event-triggering instructions inside the basic block is handled as a black box). Especially for programs consisting of a few, yet very large, basic blocks with multiple event-triggering instructions, this basic block granularity may cause significant pessimism. In order to increase the precision of the model, basic blocks which contain event-triggering instructions can be re-structured before building the model by also applying the basic block splitting already mentioned in Section 4.2.4. Thereby, each event-triggering instruction is cut into an own so-called sub basic block. This refinement does not alter the produced binary at the end of the compilation, as it only refines the number of analyzed blocks.

Figure 5.13(a) shows a basic block with two event-triggering instructions before sub basic block splitting applied. When sub basic block splitting is enabled, each instruction is cut into a single sub basic block as shown in Figure 5.13(b). As this only refines the number of events per (sub) basic block, the presented methods to build the ILP model do not have to be adapted. Yet, the distribution of event-triggering instructions inside a basic block is not handled as a black box anymore, as the location of each event-triggering instruction is explicitly modeled. On the downside, splitting a basic block into multiple sub basic blocks increases the complexity of the ILP model, as the number of required variables and constraints linearly increases with the number of (sub) basic blocks. In the worst case, each event-triggering instruction introduces two additional blocks.

Maximum Events Generated per Basic Block

Beside increasing the ILP model's precision by splitting basic blocks into smaller sub basic blocks, the number of events contributed by a single basic block on the sub-path can be potentially tightened. The general idea is that events typically have a minimum interarrival time I which describes the minimal distance in time between 2 events. E.g., the minimum interarrival time I for shared memory accesses can be

bound by the memory's access latency. Therefore, the number of events contributed by a basic block i on the sub-path can be tightened by adding the following constraint:

$$a_i^+ \leq \left\lceil \frac{z_i^+}{I} \right\rceil \tag{5.94}$$

a_i^+ is the number of events along the sub-path contributed by basic block i, whereas z_i^+ is the number of accumulated cycles required to execute i on the sub-path. Since the interarrival time I is assumed to be constant, Equation (5.94) can be easily described using linear equations (further details can be found in Appendix A). While the actual distribution of event-triggering instructions inside a basic block is still handled as a black box, it is no longer assumed that all events are executed in the very first or last cycle.

Maximum Events Generated in the Current Time Interval

Similarly to restricting the number of events generated by basic block due to the event's minimum interarrival time, the overall number of events along the sub-path can be restricted. Therefore, the following constraint is inserted:

$$a_{\text{Total}}^+ \leq \left\lceil \frac{\sum_{i \in \mathcal{B}} z_i^+}{I} \right\rceil \tag{5.95}$$

As this additional constraint only considers the overall accumulated execution time along the sub-path, it primarily tightens the event arrival function for small time interval lengths.

5.4 Extraction

The previous sections introduced the general ILP model which describes the upper event arrival function $\eta^+(\Delta t)$ (respectively the lower event arrival function $\eta^-(\Delta t)$) of a program. In case only one specific value for a single time interval length Δt is required, this can easily be obtained by simply setting the Δt constant inside the model to the required value and subsequently solving the ILP. For deriving the overall event arrival function (or in case of a periodic system up to a specific time interval length), two methods are presented in the following. This can be required, e.g., when using a system-level analysis tool (e.g., Real-Time Calculus [Wan06] or SymTA/S [Ric04]), as full event arrival functions of the tasks under analysis are needed as inputs. The first method approximates a full event arrival function efficiently by sampling the precise event arrival function with a given number of equidistant sample points. The word "sampling" is used here similar to the sampling of a generic signal, where the value of an unknown signal is measured ("sampled") at discrete points in time. This method is useful for very complex event arrival functions, where the precise extraction of the entire event arrival function would be otherwise infeasible due to a long runtime required for the extraction itself (since it requires a single ILP solving for determining a single value of the event arrival function). The second method derives the event

Algorithm 5.1 Fixed granularity extraction

Inputs: $\eta(\Delta t)$ - Model of an upper (resp. lower) event arrival function; S - Number of sample points; L - Maximum interval length; U - True if upper event arrival function, else false

Output: m - Map with the max. (resp. min.) number of event arrivals with Δt as a key

```
1:  Map m
2:  T = η(L)
3:  Δt' = 0
4:  for (Δt=0, Δt ≤ L, Δt += L/S) do
5:      n = η(Δt)
6:      if U then
7:          if Δt = 0 then
8:              m[ 0 ] = n
9:          else
10:             m[ Δt' + 1 ] = n
11:         end if
12:     else
13:         m[ Δt ] = n
14:     end if
15:     Δt' = Δt
16:     if n = T then
17:         break
18:     end if
19: end for
```

arrival function with the finest granularity, meaning each step inside the event arrival function is determined.

5.4.1 Equidistant Sampling

Algorithm 5.1 shows the procedure for deriving an approximated (yet safe) event arrival function by using a given number of equidistant samples over time. An upper event arrival function $\eta^{+\prime}(\Delta t)$ is a safe approximation of the actual upper event arrival function $\eta^{+}(\Delta t)$ if the following holds:

$$\forall \Delta t : \eta^{+\prime}(\Delta t) \geq \eta^{+}(\Delta t) \tag{5.96}$$

Similarly, a lower event arrival function $\eta^{-\prime}(\Delta t)$ is a safe approximation of the actual lower event arrival function $\eta^{-}(\Delta t)$ if the following holds:

$$\forall \Delta t : \eta^{-\prime}(\Delta t) \leq \eta^{-}(\Delta t) \tag{5.97}$$

The algorithm's inputs are the total number of sample points S to be used, the maximum interval length L and whether the ILP model represents an upper event arrival function (U is true) or a lower event arrival function (U is false). Additionally, the ILP model of the corresponding event arrival function is required as an input as well

(here simply denoted as $\eta(\Delta t)$, as it represents the *exact* event arrival function). The algorithm's output is a map with an interval length Δt as a key and the corresponding maximum number of events as the value. At first, the number of events during the entire maximum interval length L is calculated in Line 2, here denoted as T. Since any event arrival function is monotonically increasing, the number of events during the entire maximum interval length L can be used as an early termination criterion later. The loop from lines 4 to 19 increases the time interval length Δt with a step size of L/S during each iteration. In each iteration, the ILP representing the exact event arrival function is solved with setting the constant Δt inside the ILP model to its current value (cf. Line 5). In case of a lower event arrival function, the resulting objective value is simply stored in the map as the corresponding value for the current time interval length (cf. Line 13). This results in a safe approximation of the lower event arrival function: For two sample points Δt_0 and Δt_1, with $\Delta t_0 < \Delta t_1$, the following properties always hold due to the lower event arrival function being monotonically increasing:

$$\eta^-(\Delta t_0) \leq \eta^-(\Delta t_1) \tag{5.98}$$

Hence, the following statement holds as well:

$$\forall \Delta t \in [\Delta t_0, \Delta t_1) : \eta^-(\Delta t) \geq \eta^-(\Delta t_0) \tag{5.99}$$

As the approximated lower event arrival function $\eta^{-\prime}(\Delta t)$ equals $\eta^-(\Delta t_0)$ in the range $\Delta t \in [\Delta t_0, \Delta t_1)$ for two subsequent sample points Δt_0 and Δt_1, $\eta^{-\prime}(\Delta t) \leq \eta^-(\Delta t)$ always holds.

If the ILP model represents an upper event arrival function, the resulting objective value is stored as the number of events for the previous time interval length incremented by one to ensure a safe approximated event arrival function (cf. Line 10). This can easily be deduced: For two sample points Δt_0 and Δt_1, with $\Delta t_0 < \Delta t_1$, the following properties always hold due to the upper event arrival function being monotonically increasing:

$$\eta^+(\Delta t_0) \leq \eta^+(\Delta t_1) \tag{5.100}$$

Hence, the following statement holds as well:

$$\forall \Delta t \in (\Delta t_0, \Delta t_1] : \eta^+(\Delta t) \leq \eta^+(\Delta t_1) \tag{5.101}$$

As the approximated upper event arrival function $\eta^{+\prime}(\Delta t)$ equals $\eta^+(\Delta t_1)$ in the range $\Delta t \in (\Delta t_0, \Delta t_1]$ for two subsequent sample points Δt_0 and Δt_1, $\eta^{+\prime}(\Delta t) \geq \eta^+(\Delta t)$ always holds. In case the time interval Δt has a length of 0, the resulting number of events is always 0 as well, hence nothing needs to be approximated and it is directly stored into the map (cf. Line 8).

If the current number of events n is equal to the number of events T during the entire maximum interval length L, the extraction can be stopped in Line 17. Since both types of event arrival functions, lower and upper, are monotonically increasing, the rest of the event arrival function is constant once n reaches T.

Figure 5.14 illustrates Algorithm 5.1. The figure depicts an actual upper event arrival function $\eta^+(\Delta t)$ and a lower one $\eta^-(\Delta t)$. These event arrival functions are

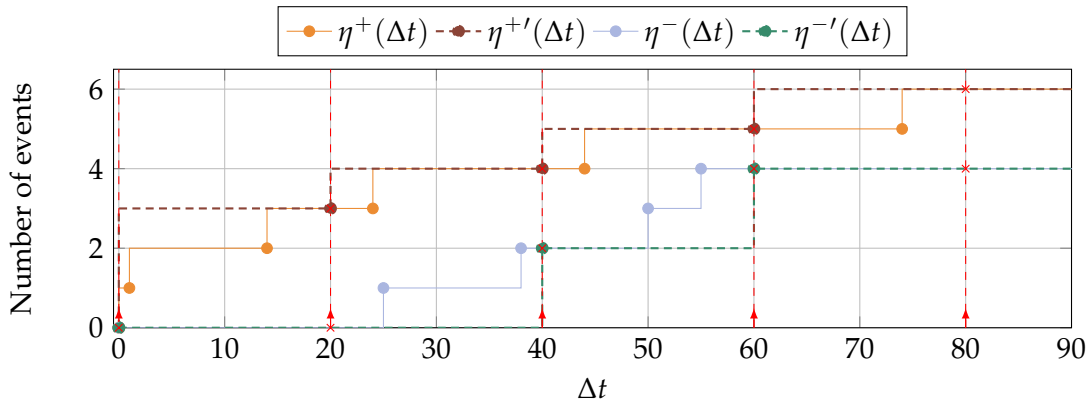


Figure 5.14. – An Upper event arrival function $\eta^+(\Delta t)$ and a lower event arrival function $\eta^-(\Delta t)$ depicted with their sampled approximations.

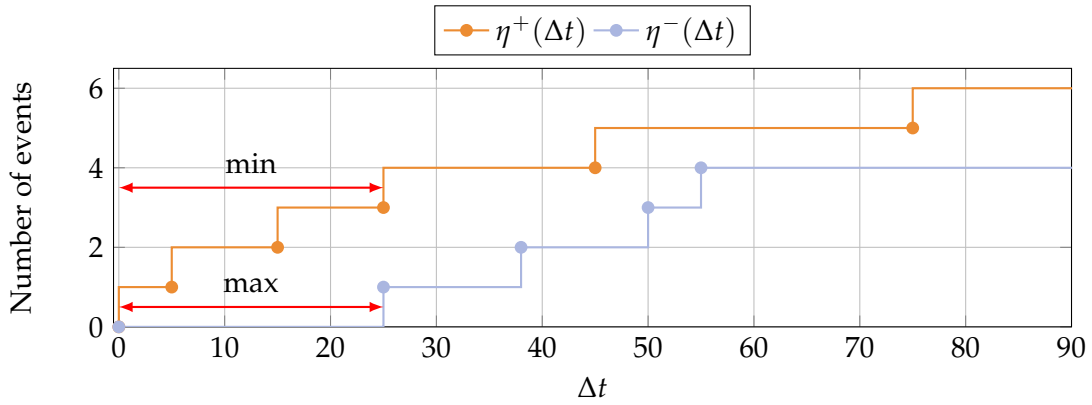


Figure 5.15. – Depiction of how a point of discontinuity in an upper (resp. lower) event arrival function corresponds to the minimum (resp. maximum) time interval in which a certain amount of events are generated.

sampled using 5 sample points with a distance of 20 time units between each sample point. The upward facing red arrows depict the time interval lengths for which a sample point is calculated. $\eta^{+\prime}(\Delta t)$ is the derived sampled upper event arrival function, whereas $\eta^{-\prime}(\Delta t)$ is the lower event arrival function derived from the samples.

5.4.2 Exact Extraction

The procedure discussed in the previous section to derive an approximated event arrival function could be used as well to derive an exact function (meaning that each step is precisely calculated) by using a very large number of samples ($S = L$), essentially solving the ILP model for each possible value of Δt. Yet, this would result in an excessive amount of ILPs to solve even for comparably small programs.

The flexible description of the ILP model allows a far more efficient way of deriving the exact event arrival function with a near minimal number of ILPs to solve. Instead of solving the ILP model for every single possible value of Δt, Δt is specifically derived for every point of discontinuity inside the event arrival function. Figure 5.15 illustrates this idea. For deriving the exact event arrival function, the time interval

Algorithm 5.2 Exact Extraction

Inputs: $\eta(\Delta t)$ - Model of an upper (resp. lower) event arrival function; L - Maximum interval length; U - True if upper event arrival function, else false
Output: m - Map with the max. (resp. min.) number of event arrivals with Δt as a key

```
1:  Map m
2:  m[ 0 ] = 0
3:  X = η(L)
4:  δ(n) = adaptILP()
5:  for (n = 0, n ≤ X, ++n) do
6:      Δt = δ(n)
7:      if U then
8:          m[ Δt ] = n
9:      else
10:         m[ Δt + 1 ] = n + 1
11:     end if
12: end for
```

length values Δt for each point of discontinuity need to be known, as the function is otherwise constant. For an upper event arrival function, this relates to finding the shortest sub-path which generates a specific number of events. E.g., as shown in Figure 5.15, the step to 4 events can be found along that sub-path which generates this number of events and takes the minimal amount of cycles. Similarly, for a lower event arrival function, this relates to finding the longest sub-path which generates a specific number of events.

In general, this concept underlines the correlation of an upper event arrival function $\eta^+(\Delta t)$ and its corresponding lower distance function $\delta^-(n)$, as well as $\eta^-(\Delta t)$ and its corresponding $\delta^+(n)$. This is due to the fact that, strictly speaking, the ILP model is in this case no longer used to calculate an event arrival function, but the corresponding distance function. For this, an additional constraint is added and the objective function of an upper event arrival function ILP (cf. the original Equation (5.13)) is replaced as follows:

$$a^+_{\text{Total}} = A \tag{5.102}$$
$$\min : \Delta t \tag{5.103}$$

A represents the number of events to be generated along the sub-path. Δt is no longer a constant here, but an ILP variable. The same constraint is added for a lower event arrival function ILP and also its objective function (cf. the original Equation (5.21)) is replaced as follows:

$$a^-_{\text{Total}} = A \tag{5.104}$$
$$\max : \Delta t \tag{5.105}$$

Algorithm 5.2 shows how to derive an exact event arrival function up to a maximum interval length L. The initial ILP description of Algorithm 5.2 (cf. Line 3) *does not* feature the previously described modifications of the objective function and the

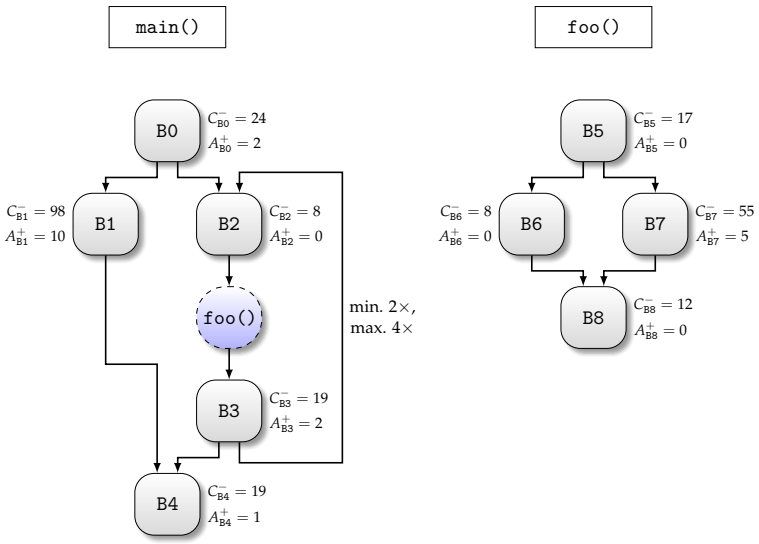


Figure 5.16. – Exemplary control-flow graph with annotated best-case execution times and numbers of events.

additional number of events setting constraint. The maximum (resp. minimum) number of events in the overall maximum time interval of length L is calculated in Line 3. The modifications from Equations (5.102) and (5.103) for an upper event arrival function (resp. Equations (5.104) and (5.105) for a lower event arrival function) are done in Line 4. As mentioned before, these modifications essentially turn the ILP model into a distance function $\delta(n)$. The for-loop from Lines 5 to 12 iterates over each possible number of events (the points of discontinuity in the event arrival function) and derives the minimum (resp. maximum) sub-path length to generate these events. For an event arrival function with n points of discontinuity in a given time interval Δt, only $n+1$ ILPs need to be solved (the first one to derive the maximum number of events during the whole time interval Δt), resulting in a near minimal number of ILPs to be solved overall (as with each ILP solving only a single value of an event arrival function $\eta(\Delta t)$ can be derived).

5.5 An Illustrative Example

In the following, the basic ILP model for deriving an upper event arrival function $\eta^+(\Delta t)$ is set up for a given, non-periodic program. Figure 5.16 shows the control-flow graph of the program. Each basic block is annotated with its best-case execution time C^- and its maximum number of generated events A^+. The `main`-function has a tail-controlled loop with an upper loop bound of 5 and a lower loop bound of 3 (note that the back-edge is executed one time less than the loop body due to the loop being tail-controlled).

At first, the node law for the program's entrypoint basic block `B0` is created:

$$p_{\ominus,\mathtt{B0}} - e_{\mathtt{B0}} = p_{\mathtt{B0,B1}} - s_{\mathtt{B0,B1}} + p_{\mathtt{B0,B2}} - s_{\mathtt{B0,B2}} \tag{5.106}$$

As basic block B0 is the program's entrypoint, a virtual source $\ominus$ is inserted as predecessor. In this case, the starting edge indicator $s_{\ominus,\mathtt{B0}}$ for basic block B0 is equal to its only regular incoming edge $p_{\ominus,\mathtt{B0}}$:

$$s_{\ominus,\mathtt{B0}} = p_{\ominus,\mathtt{B0}} \tag{5.107}$$

Subsequently, the node laws for the remaining blocks of the `main`-function are inserted:

$$p_{\mathtt{B0},\mathtt{B1}} - e_{\mathtt{B1}} = p_{\mathtt{B1},\mathtt{B4}} - s_{\mathtt{B1},\mathtt{B4}} \tag{5.108}$$
$$p_{\mathtt{B0},\mathtt{B2}} + p_{\mathtt{B3},\mathtt{B2}} - e_{\mathtt{B2}} = p_{\mathtt{B2},\mathtt{B5}} - s_{\mathtt{B2},\mathtt{B5}} \tag{5.109}$$
$$p_{\mathtt{B8},\mathtt{B3}} - e_{\mathtt{B3}} = p_{\mathtt{B3},\mathtt{B4}} - s_{\mathtt{B3},\mathtt{B4}} + p_{\mathtt{B3},\mathtt{B2}} - s_{\mathtt{B3},\mathtt{B2}} \tag{5.110}$$
$$p_{\mathtt{B1},\mathtt{B4}} + p_{\mathtt{B3},\mathtt{B4}} - e_{\mathtt{B4}} = 0 \tag{5.111}$$

Note that there are no outgoing flows allowed for the program's end block B4. Similarly, the node laws for the function `foo` are inserted into the ILP:

$$p_{\mathtt{B2},\mathtt{B5}} - e_{\mathtt{B5}} = p_{\mathtt{B5},\mathtt{B6}} - s_{\mathtt{B5},\mathtt{B6}} + p_{\mathtt{B5},\mathtt{B7}} - s_{\mathtt{B5},\mathtt{B7}} \tag{5.112}$$
$$p_{\mathtt{B5},\mathtt{B6}} - e_{\mathtt{B6}} = p_{\mathtt{B6},\mathtt{B8}} - s_{\mathtt{B6},\mathtt{B8}} \tag{5.113}$$
$$p_{\mathtt{B5},\mathtt{B7}} - e_{\mathtt{B7}} = p_{\mathtt{B7},\mathtt{B8}} - s_{\mathtt{B7},\mathtt{B8}} \tag{5.114}$$
$$p_{\mathtt{B6},\mathtt{B8}} + p_{\mathtt{B7},\mathtt{B8}} - e_{\mathtt{B8}} = p_{\mathtt{B8},\mathtt{B3}} - s_{\mathtt{B8},\mathtt{B3}} \tag{5.115}$$

Afterwards, the timing limiting constraints for the basic blocks are added:

$$z^+_{\mathtt{B0}} \geq 24 \cdot p_{\ominus,\mathtt{B0}} - 23 \cdot b_{\mathtt{B0}} \tag{5.116}$$
$$z^+_{\mathtt{B1}} \geq 98 \cdot p_{\mathtt{B0},\mathtt{B1}} - 97 \cdot b_{\mathtt{B1}} \tag{5.117}$$
$$z^+_{\mathtt{B2}} \geq 8 \cdot (p_{\mathtt{B0},\mathtt{B2}} + p_{\mathtt{B3},\mathtt{B2}}) - 7 \cdot b_{\mathtt{B2}} \tag{5.118}$$
$$z^+_{\mathtt{B3}} \geq 19 \cdot p_{\mathtt{B8},\mathtt{B3}} - 18 \cdot b_{\mathtt{B3}} \tag{5.119}$$
$$z^+_{\mathtt{B4}} \geq 19 \cdot (p_{\mathtt{B3},\mathtt{B4}} + p_{\mathtt{B1},\mathtt{B4}}) - 18 \cdot b_{\mathtt{B4}} \tag{5.120}$$
$$z^+_{\mathtt{B5}} \geq 17 \cdot p_{\mathtt{B2},\mathtt{B5}} - 16 \cdot b_{\mathtt{B5}} \tag{5.121}$$
$$z^+_{\mathtt{B6}} \geq 8 \cdot p_{\mathtt{B5},\mathtt{B6}} - 7 \cdot b_{\mathtt{B6}} \tag{5.122}$$
$$z^+_{\mathtt{B7}} \geq 55 \cdot p_{\mathtt{B5},\mathtt{B7}} - 54 \cdot b_{\mathtt{B7}} \tag{5.123}$$
$$z^+_{\mathtt{B8}} \geq 12 \cdot (p_{\mathtt{B6},\mathtt{B8}} + p_{\mathtt{B7},\mathtt{B8}}) - 11 \cdot b_{\mathtt{B8}} \tag{5.124}$$

And the event limiting constraints as well:

$$a^+_{\mathtt{B0}} \leq 2 \cdot p_{\ominus,\mathtt{B0}} \tag{5.125}$$
$$a^+_{\mathtt{B1}} \leq 10 \cdot p_{\mathtt{B0},\mathtt{B1}} \tag{5.126}$$
$$a^+_{\mathtt{B2}} \leq 0 \cdot (p_{\mathtt{B0},\mathtt{B2}} + p_{\mathtt{B3},\mathtt{B2}}) \tag{5.127}$$
$$a^+_{\mathtt{B3}} \leq 2 \cdot p_{\mathtt{B8},\mathtt{B3}} \tag{5.128}$$
$$a^+_{\mathtt{B4}} \leq 1 \cdot (p_{\mathtt{B1},\mathtt{B4}} + p_{\mathtt{B3},\mathtt{B4}}) \tag{5.129}$$
$$a^+_{\mathtt{B5}} \leq 0 \cdot p_{\mathtt{B2},\mathtt{B5}} \tag{5.130}$$
$$a^+_{\mathtt{B6}} \leq 0 \cdot p_{\mathtt{B5},\mathtt{B6}} \tag{5.131}$$
$$a^+_{\mathtt{B7}} \leq 5 \cdot p_{\mathtt{B5},\mathtt{B7}} \tag{5.132}$$
$$a^+_{\mathtt{B8}} \leq 0 \cdot (p_{\mathtt{B6},\mathtt{B8}} + p_{\mathtt{B7},\mathtt{B8}}) \tag{5.133}$$

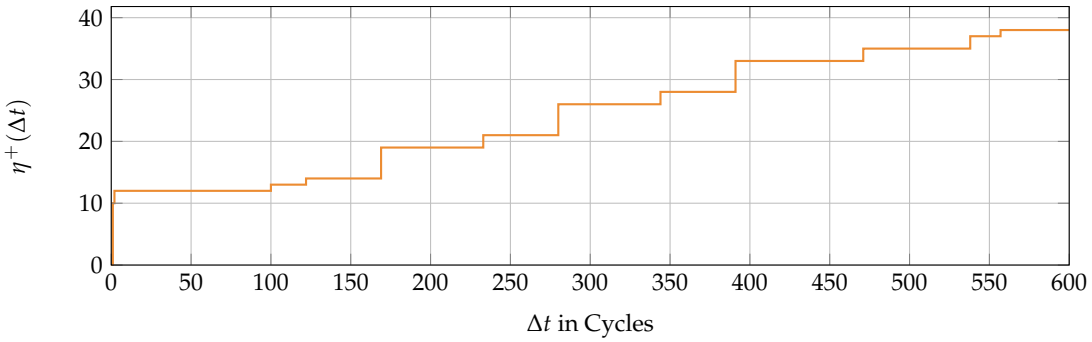


Figure 5.17. – Upper event arrival function $\eta^+(\Delta t)$ from the exemplary control-flow graph depicted in Figure 5.16.

All events generated along the sub-path are accumulated:

$$a^+_{\text{Total}} = a^+_{\texttt{B0}} + a^+_{\texttt{B1}} + \ldots + a^+_{\texttt{B8}} \tag{5.134}$$

Subsequently, the constraints to limit the maximum flow through the tail-controlled loop of function `main` are inserted:

$$p_{\texttt{B3,B2}} \leq n^T_\ell \tag{5.135}$$
$$n^T_\ell = 4 \cdot (p_{\texttt{B0,B2}} + s_\ell) \tag{5.136}$$
$$s_\ell = s_{\texttt{B3,B2}} \vee s_{\texttt{B8,B3}} \vee s_{\texttt{foo}} \tag{5.137}$$

Also the constraint to enforce the lower loop bound is inserted:

$$p_{\texttt{B3,B2}} \geq 2 \cdot \min(p_{\texttt{B0,B2}}, p_{\texttt{B3,B4}}) \tag{5.138}$$

As the function `foo` is only called from one function, the constraints enforcing fitting call- and return-pairs are simple:

$$p_{\texttt{B2,B5}} \geq p_{\texttt{B8,B3}} - s_{\texttt{foo}} \tag{5.139}$$
$$p_{\texttt{B8,B3}} \geq p_{\texttt{B2,B5}} - e_{\texttt{foo}} \tag{5.140}$$

The variable $p_{\texttt{B2,B5}}$ here represents the calling-edge, whereas $p_{\texttt{B8,B3}}$ corresponds to the return edge.

Eventually, the constraint limiting the total length of the sub-path and the objective are inserted:

$$\Delta t \geq z^+_{\texttt{B0}} + z^+_{\texttt{B1}} + \ldots + z^+_{\texttt{B8}} \tag{5.141}$$
$$\max : a^+_{\text{Total}} \tag{5.142}$$

The complete upper event arrival function $\eta^+(\Delta t)$ resulting from this model is depicted in Figure 5.17 (derived using the exact extraction method from Section 5.4.2). The first step to 10 events is due to the basic block `B1`'s maximum number of 10 generated events (note that this reflects the basic model without the additional constraints to limit the minimum time between events). Immediately after, the function

steps to 12 events which is caused by the sub-path {B0, B1}. At a time interval length of 100 cycles, a step to 13 events occurs. This corresponds to the sub-path {B0, B1, B4}. For the following step to 14 events, the loop becomes part of the sub-path {B7, B8, B3, B2, B5, B6, B8, B3, B2, B5, B7}. The shortest sub-path for reaching 19 events is the same as for 14 events, yet executing the basic block B7 instead of B6 during the second execution of function foo(), leading to {B7, B8, B3, B2, B5, B7, B8, B3, B2, B5, B7}. The next steps to 21, 26, 28, 33, and 35 events follow a repetitive pattern where each new iteration of the loop can either add 2 new events by including an additional execution of B3, or 5 additional events for each further execution of B7. For the second to last step to 37 events, the sub-path starts at B0 and ends at B3 after executing the loop 5 times. Finally, the event arrival function converges to 38 events at a time interval length of 557 cycles with the additional execution of basic block B4.

5.6 Sensitivity Analysis

This section evaluates the influence of variable parameters of the presented event arrival function extraction on timing analyses. Therefore, the following part introduces the timing analysis method used for the evaluation, Section 5.6.2 details the overall evaluation setup and Sections 5.6.3 to 5.6.5 present the actual results.

5.6.1 Timing Analysis

Event arrival functions are commonly used within system-level compositional timing analyses (cf. Section 2.2) to describe task activations or shared resource access behavior. These types of analyses typically handle tasks as an abstract tuple of metrics, such as their WCET, their activation pattern or their shared resource access behavior. By analyzing the competing resource accesses (or task activations in case of a multi-task system), the analysis is able to derive an upper bound on the waiting time due to stalling for each task in the system. The compositional timing analysis then adds this derived upper bound of induced waiting times due to resource conflicts onto the task's WCET to determine the task's WCRT.

As this is a common use-case for event arrival functions, a system-level compositional timing analysis approach is used for evaluating the influence of the different parameters of the presented event arrival function extraction. Since a single task is assumed to be allocated onto each core in the system, the timing analysis only needs to derive a safe upper bound on the bus stall times for each task, as a task cannot be preempted by another one. The WCRT R_i^+ of a task i in a multi-core system using a fixed priority-based bus arbitration is derived using the following equation:

$$R_i^+ = C_i^+ + \underbrace{\sum_{j \in \mathcal{H}_i} \eta_j^{+\prime}(R_i^+) \cdot F_S}_{X} + \underbrace{\min\left(\sum_{j \in \mathcal{U}_i} \eta_j^{+\prime}(R_i^+), A_{E,i}^+\right) \cdot (F_S - 1)}_{Y} \tag{5.143}$$

The WCET of task i is denoted by C_i^+ (which does not include any interferences of any other cores), whereas F_S denotes the net access latency of the shared memory. The set $\mathcal{H}_i$ contains all tasks allocated onto cores with a higher bus priority than

the core on which task i is executed on, while the set $\mathcal{U}_i$ contains all tasks on lower priority cores. $\eta_i^{+\prime}(\Delta t)$ is the approximated event arrival function of task i. The maximum number of bus accesses during one complete execution of task i is denoted as $A^+_{\mathrm{E},i}$. Equation (5.143) is very similar to the analysis of worst-case response times for fixed priority tasks initially given by Joseph and Pandya [JP86] and later refined and extended by, e.g., Lehoczky [Leh90] and others. The WCRT is derived by adding to its WCET C_i^+ an upper bound on waiting times due to bus contention, marked in Equation (5.143) as two parts, X and Y. The first part X describes the maximum bus waiting time of task i due to higher priority cores. In the worst case, each access of the higher priority cores is blocking task i from performing its shared memory accesses. Additionally, each access of task i could be blocked by up to $F_{\mathrm{S}} - 1$ cycles by a lower priority core in case it received a bus grant before task i requests it. This is represented by the second part, marked as Y. Since the total number of shared memory accesses performed by task i during one execution is limited by $A^+_{\mathrm{E},i}$, not more accesses can be blocked. Similarly, the maximum number of shared memory accesses initiated by lower priority cores during one execution of task i is limited by $\sum_{j \in \mathcal{U}_i} \eta_j^{+\prime}(R_i^+)$. Therefore, the maximum number of accesses of task i that can be blocked due to lower priority cores is limited by the minimum of these two values. Equation (5.143) is solved iteratively, using the WCET C_i^+ of task i as an initial value of its WCRT R_i^+ until a stable value is reached.

In case a multi-core system has a round robin-based bus arbitration, the following equation is used to derive the WCRT R_i^+ of a task i:

$$R_i^+ = C_i^+ + \sum_{j \in \Gamma, j \neq i} \min\left(\eta_j^{+\prime}(R_i^+), A^+_{\mathrm{E},i}\right) \cdot F_{\mathrm{S}} \tag{5.144}$$

As there are no fixed priorities in a round robin-based bus arbitration, any access of task i may be blocked by another access of any other core. Yet, as the priorities are rotated after each access, the maximum number of accesses of task i being blocked by another task j (which is on a different core) is limited by $\min(\eta_j^{+\prime}(R_i^+), A^+_{\mathrm{E},i})$. As task i does not initiate more than $A^+_{\mathrm{E},i}$ accesses, no more accesses may be blocked due to any other core. As the concurrent task j does not initiate more than $\eta_j^{+\prime}(R_i^+)$ accesses during the execution time of task i, also not more accesses may be blocked. Each access blocked by another core may results in a worst-case blocking time of F_{S} cycles.

Timing analyses for multi-core systems with a TDMA-based bus arbitration are not discussed here, as they can not take advantage of a detailed knowledge of the access distribution using event arrival functions, but rather need an offset analysis as presented by Kelter [Kel15].

5.6.2 Setup

For the evaluation, the exemplary base multi-core architecture shown in Figure 2.6 (cf. Page 13) with shared memories as well as private SPMs for each core is assumed. Different architectures with 2, 4 and 8 cores are evaluated. In case of a fixed priority-based bus arbitration, it is assumed that core 0 has the lowest priority and core $n-1$ the highest. It is assumed that the code of each task is allocated to the corresponding core's private instruction SPM (which is assumed to be sufficiently sized), whereas all

data objects of the task are placed in the shared data memory. These assumptions are done in order to see a meaningful influence of the event arrival function extraction parameters on the tasks' WCRTs without performing an SPM allocation optimization beforehand. Two different bus arbitration methods are evaluated: A fixed priority-based arbitration, as well as a round robin-based one. These two bus arbitration methods are chosen as they are the pre-dominant ones in modern multi-core COTS, and since a precise worst-case timing analysis of them is comparably difficult [Kel15], as they are work-conserving. The shared memory is assumed to have an access latency of 6 cycles (similar to the default setting of existing embedded systems [NXP09a, Inf07]), whereas an SPM access is assumed with a single cycle latency. Each task has a periodic activation where the task's period is set to two times its WCET (leading to a utilization of 50% per core in isolation). Benchmarks of the following benchmark suites were evaluated: MRTC [GBEL10], MediaBench [LPM97], StreamIt [Str18], and UTDSP [LCS92]. Additionally, a set of miscellaneous benchmarks, mostly consisting of de- and encoders were evaluated as well. Benchmarks from this set were randomly ordered and grouped by using a sliding window over the list. This results in a total number of 86 different systems to be evaluated for each multi-core setting (cf. Appendix E.2 for a detailed overview of the different benchmark sets).

For each system, 6 different configurations were evaluated for the event arrival function extraction: The number of sample points used for deriving an approximated event arrival function per task was set to 100, 500 or 1 000. The samples are taken equidistantly. The WCET of each task is used as the maximum length L of the time interval for deriving the approximated event arrival function (cf. Algorithm 5.1). Orthogonal to this, a fine granular model (with all refinement methods described in Section 5.3.7) or a more coarse one (without the refinement methods) is generated. Deriving exact event arrival functions for each task (as described in Section 5.4.2) was tested, yet not included in this evaluation, as it exceeded the set timeout limit in most cases. Overall, this results in 3 096 different evaluated configurations (dual-, quad- and octacore-systems; 86 benchmark sets each; 6 different extraction settings; round robin and fixed priority bus arbitration).

The timing analysis is carried out by first deriving each task's WCET using the internal WCET analyzer provided by the WCET-aware C compiler (WCC) [FL10] with its value analysis being extended by using aiT's value analyzer (version 18.10) for a more precise classification of shared memory accesses. Subsequently, an estimated event arrival function is extracted for each task using the presented methods and according to the parameters chosen. Finally, the WCRT of each task is derived in a compositional manner by applying the equations presented in the previous section. All experiments were performed on an Intel Xeon Server (48 cores at 3.2 GHz with 1.48 TB RAM) and ILPs were solved using Gurobi 8.1.0, whereas each ILP solving process was limited to 1 thread. All benchmarks were compiled with several ACET-oriented optimizations activated (`-O2` optimization flag of the WCC compiler, cf. Chapter 3). A general timeout of 1 h per core in the evaluated system is used. I.e., the evaluation of a quad-core system has a timeout of 4 h.

5.6.3 Dual-Core Evaluation

Figure 5.18 shows the evaluation results for systems with 2 cores and a fixed priority-based bus arbitration. The different event arrival function extraction configurations

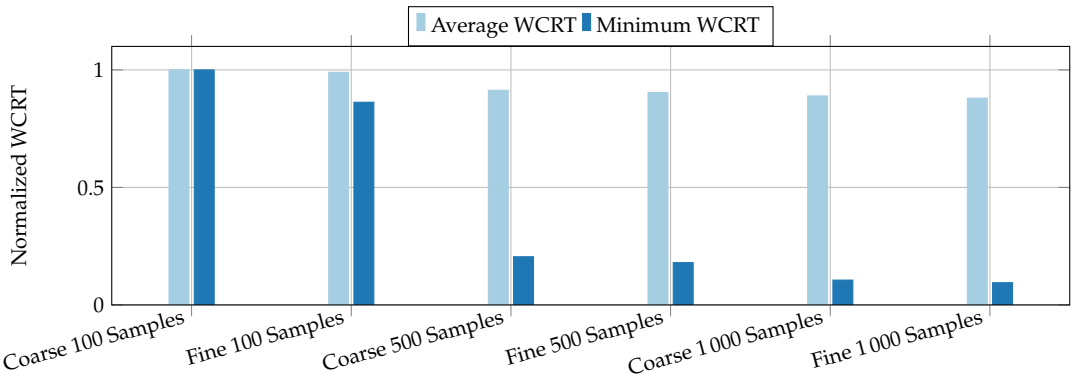


Figure 5.18. – Normalized average and minimum WCRTs among all tasks evaluated for dual-core systems with a fixed priority-based bus arbitration depending on the event arrival function extraction parameters.

are marked on the x-axis. The y-axis denotes the WCRT, normalized to the most coarse configuration: 100 samples and no refinements applied. For each configuration, the average (geometric mean) of all normalized WCRTs of this configuration, as well as the lowest normalized WCRT of this configuration is shown. E.g., the normalized WCRT of a task when using the refined ILP model and 100 sample points ("Fine 100 Samples") for deriving each approximated event arrival function in the system is on average 0.99, meaning the additional refinements reduce the WCRT on average by 1%. Similarly, the minimum normalized WCRT is 0.86 for this case, meaning the largest WCRT reduction compared to the most coarse configuration is 14%.

On average, increasing the number of sample points from 100 to 500 results in a WCRT reduction of around 9%, independent of whether the coarse or fine ILP model was used. A further increase of sample points from 500 to 1 000 results on average in an additional reduction of around 3%. Regarding the overall largest achievable WCRT reductions, an immense step from using 100 to 500 sample points can be seen as well. By using 500 instead of 100 sample points, the WCRT estimation can be reduced by up to 79% for the coarse ILP model and up to 81% for the fine ILP model.

The additional effects when using the refined ILP model compared against the coarse model are more subtle, yet existing. On average, the refined ILP model improves the WCRT by 1% when using the same number of sample points: For 100 samples the average normalized WCRT is 1.0 without refinements and 0.99 with refinements, at 500 samples the average normalized WCRT is 0.91 without refinements and 0.90 with refinements and for 1 000 samples the average normalized WCRT is 0.89 without refinements and 0.88 with refinements.

The results for dual-core systems using a round robin-based arbitration are shown in Figure 5.19. The structure of the graph is identical to the previous one, yet note the y-axis' lower bound of 0.8. As can be seen already from the different y-axis scaling, the effects on the WCRT due to the different event arrival function parameters are considerably lower for the round robin-based bus arbitration than for the fixed priority one. Neither a larger number of sample points, nor using a refined model significantly reduces the average WCRT estimation. This lower effect on the WCRT estimation is due to the inherent upper bound on blocked cycles due to a bus accesses of the round robin-based bus arbitration. While potentially *all* accesses of higher

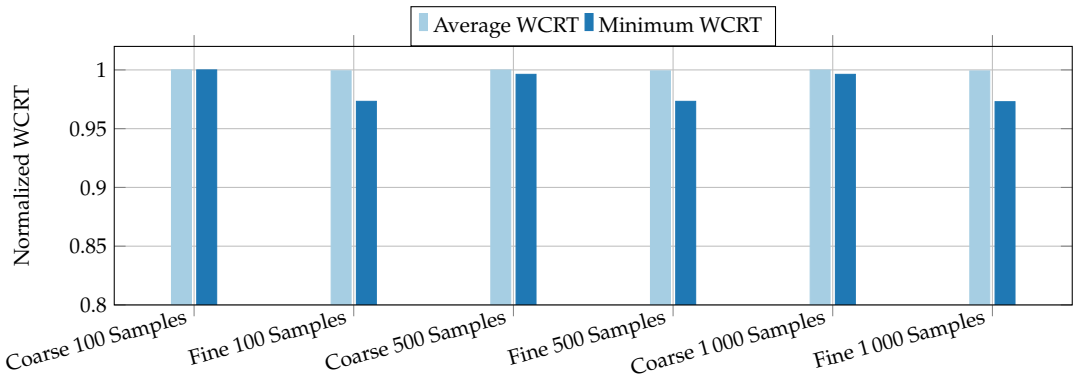


Figure 5.19. – Normalized average and minimum WCRTs among all tasks evaluated for dual-core systems with a round robin-based bus arbitration depending on the event arrival function extraction parameters.

priority cores may win arbitration against a single access of a lower priority core, the worst-case blocking time for a single access is tightly bounded in case of a round robin-based arbitration. This is reflected in the corresponding Equation (5.144) to derive an upper bound on the total number of cycles a task may be blocked due to concurrent accesses: Due to the min-term, the estimated WCRT of a task is way less dependent on the accuracy of the derived event arrival functions of the tasks on concurrent cores.

Regarding the largest WCRT reductions achieved, it is noticeable that using the fine-grained ILP model seems to result in a larger effect compared to an increase in sample points for the system evaluated: While the minimum normalized WCRT observed when using the refined ILP model with 100 sample points is 0.97, it is 0.99 when using the coarse ILP model with 500 samples. Even when increasing the number of sample points to 1 000, the minimum normalized WCRT is still 0.99 when using the coarse ILP model. This also shows the detail, that the number of used sample points and the ILP model's granularity are somewhat orthogonal parameters. E.g., in case of a mostly sequential program with a lot of shared memory accesses, only increasing the number of sample points will most likely not increase the timing analysis precision (as only a few basic blocks contain all shared memory accesses). By using the ILP model with all refinements integrated, a WCRT reduction of up to 3% can be seen among all dual-core systems with a round robin-based bus arbitration.

5.6.4 Quad-Core Evaluation

Figure 5.20 shows the evaluation results for multi-core systems with 4 cores and a fixed priority-based bus arbitration. The results are similar to the corresponding dual-core system evaluation, yet with increased WCRT reductions for the different parameter settings. Using 500 instead of 100 sample points with the coarse ILP model reduces the WCRT estimation by around 12% on average, whereas using 1 000 sample points decreases it further by 3.7%. The fine-grained ILP model yields an WCRT reduction of around 5.5% on average when using the same number of sample points: The average normalized WCRT using the coarse ILP model and 100 samples is 1.0, whereas using the refined model and 100 samples reduces the average normalized

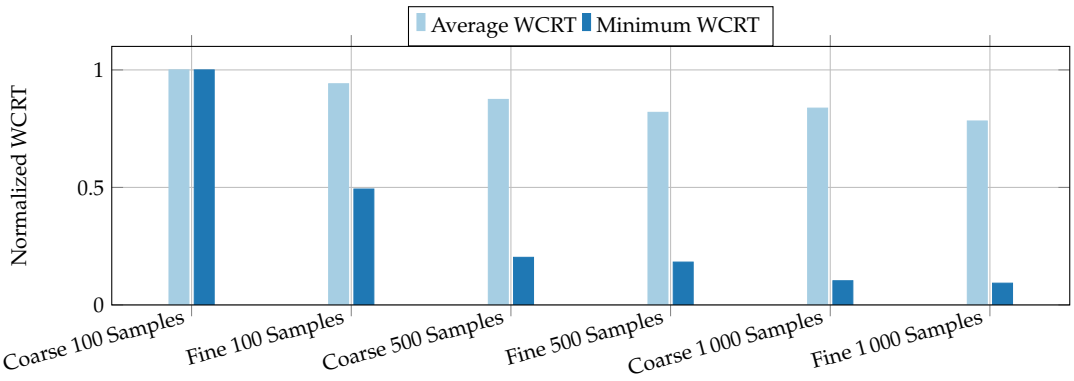


Figure 5.20. – Normalized average and minimum WCRTs among all tasks evaluated for quad-core systems with a fixed priority-based bus arbitration depending on the event arrival function extraction parameters.

WCRT to 0.94. Using 500 samples and the coarse ILP model results in an average normalized WCRT of 0.87, while using the refined ILP model with the same number of samples reduces the average normalized WCRT to 0.82. Similarly, the average normalized WCRT decreases from 0.84 to 0.78 when using 1 000 sample points and switching to the refined ILP model.

Similarly, also the largest WCRT reduction values observed per setting improved for the quad-core systems. When using the fine-grained ILP model and 100 sample points, a WCRT reduction of up to 50% compared to the coarse model can be noticed. For the coarse ILP model combined with 500 sample points, a reduction of up to 80% against the baseline is observed. Combining the fine-grained ILP model with an increased number of sample points further increases the observed maximum WCRT reductions. Overall, the largest WCRT reduction can be seen for the fine-grained ILP model with 1 000 sample points, leading to a reduction of more than 90%. These substantial WCRT reductions can be explained by the worst-case behavior of the fixed priority bus arbitration and the increased number of cores. With a higher number of cores, the worst-case blocking time for shared memory accesses increases, leading to an immediate WCRT reduction with every potential competing bus access less due to a more a precise analysis.

The evaluation results for the quad-core systems evaluated with a round robin-based bus arbitration are shown in Figure 5.21. Note the y-axis' lower bound of 0.8. Similar to the previous evaluation for dual-core systems, the event arrival function extraction parameters hardly have any influence on the estimated WCRT values when using a round robin-based bus arbitration. On average, the WCRT does not significantly change if a higher number of sample points or the fine-grained ILP model is used. Yet, as seen in the dual-core evaluation, for a very few tasks, the WCRT can be reduced by using the fine-grained ILP model. Here, a WCRT reduction of up to 2.4% can be achieved by using the refined ILP model. This relative value is lower than compared to the maximum WCRT reduction observed for the dual-core systems, as only very few tasks can benefit from it. The worst-case number of blocked cycles during a bus access of a single task is derived for each competing core individually (see the min-term in Equation (5.144)). Therefore, the relative WCRT

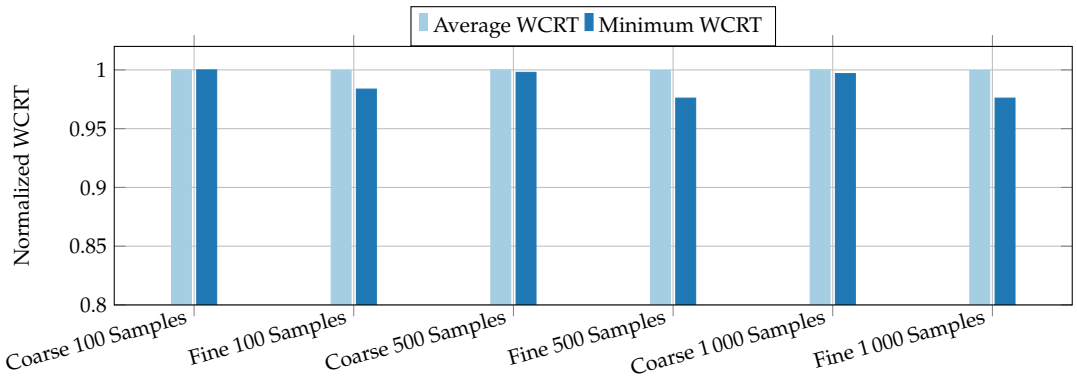


Figure 5.21. – Normalized average and minimum WCRTs among all tasks evaluated for quad-core systems with a round robin-based bus arbitration depending on the event arrival function extraction parameters.

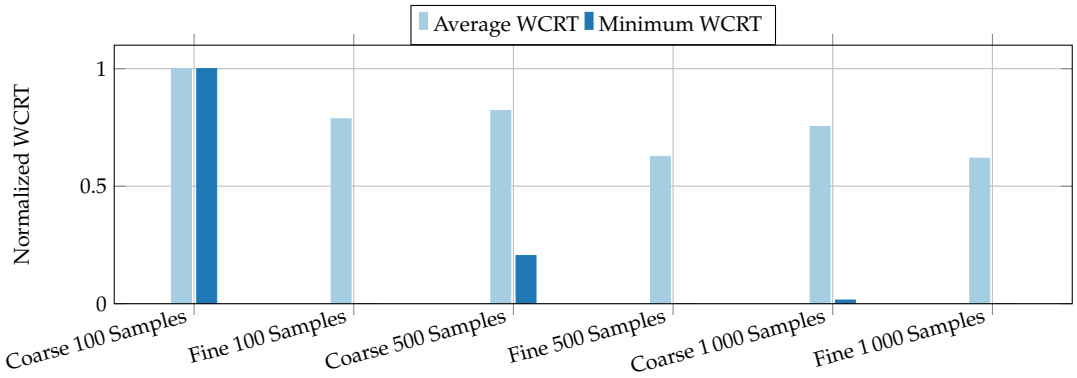


Figure 5.22. – Normalized average and minimum WCRTs among all tasks evaluated for octa-core systems with a fixed priority-based bus arbitration depending on the event arrival function extraction parameters.

reduction decreases in comparison, if the ratio of tasks benefiting from the improved event arrival function does not increase as well with the number of cores.

5.6.5 Octa-Core Evaluation

Figure 5.22 shows the evaluation results for the octa-core systems with a fixed priority-based bus arbitration. The results follow the insights of the previous dual-core and quad-core systems, yet further amplified due to the additional number of cores. On average, the normalized WCRT decreases from 1.0 to 0.822 when using 500 instead of 100 sample points with the coarse-grained ILP model, which corresponds to an average WCRT reduction of 17.8%. Increasing the number of sample points further to 1 000 lowers the average normalized WCRT to 0.754. The refined model reduces the average normalized WCRT to 0.787 when using 100 samples, to 0.626 when using 500 samples and down to 0.619 with 1 000 samples. On average, the WCRT is 18.1% lower when using the refined ILP model with the same number of samples.

The maximum WCRT reductions observed for the octa-core evaluation are immense. By using the refined ILP model and 100 sample points, the largest WCRT reduction

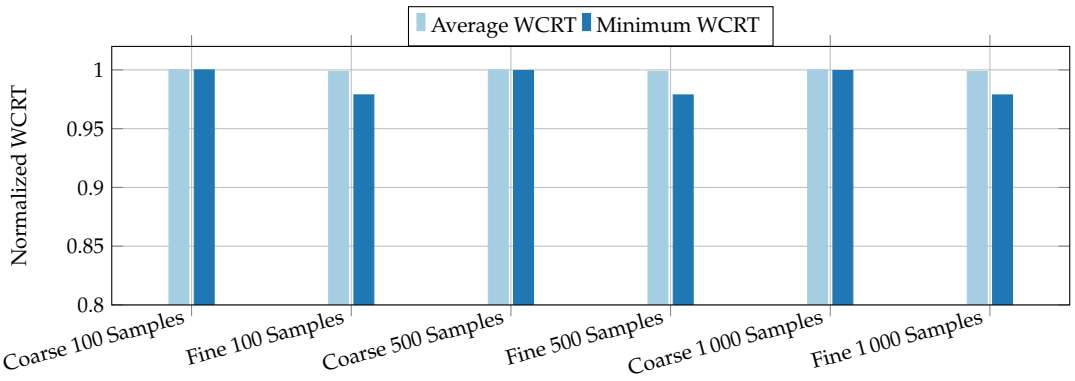


Figure 5.23. – Normalized average and minimum WCRTs among all tasks evaluated for octa-core systems with a round robin-based bus arbitration depending on the event arrival function extraction parameters.

compared to the baseline observed is more than 99%. In this particular case, a very data intensive task is allocated to one of the cores with the lowest bus priorities. Therefore, the vast majority of the task's WCRT is determined only by the worst-case blocking times of higher priority cores. By using the refined ILP model, the number of potentially competing accesses is reduced significantly in this case, and therefore also its WCRT. A similar trend can be seen when using 500 instead of 100 sample points with the coarse ILP model, yet with a lower maximum reduction. Here, the WCRT is reduced by up to 79.5% compared to the baseline. When increasing the number of samples to 1 000 in conjunction with the coarse ILP model, the maximum WCRT reduction observed increases to 98.4%.

The results for multi-core systems with 8 cores and a round robin-based bus arbitration are shown in Figure 5.23. Similar as before, the different parameters of the event arrival function generation have hardly any impact on the derived average WCRT values. The maximum WCRT reductions observed follow the same trend as for the dual- and quad-core systems, as only the refined ILP model seems to have any impact if at all. As seen for quad-core systems, the relative maximum WCRT reduction observed for the octa-core systems of 2.1% is slightly lower than compared to the dual- or quad-core systems. Similarly, this can be explained by the very few tasks affected at all: As the proportion of tasks' WCRTs influenced by the event arrival function generation parameters is decreasing per system (since the number of cores is increasing), also the relative WCRT reduction is decreasing.

5.6.6 Runtime

The average runtime (arithmetic mean) required for the event arrival function generation and following timing analysis depending on the ILP model and the number of used samples is depicted in Figure 5.24. The x-axis lists the different configurations (namely ILP model and number of samples), whereas the y-axis displays the average runtime in seconds. As expected, the required runtime increases with the number of samples and the use of the refined ILP model. Whereas the average runtime for the coarse ILP model and 100 samples is at 172 s, it increases to 2 211 s on average when using the same number of sample points but the refined model. On average, it takes

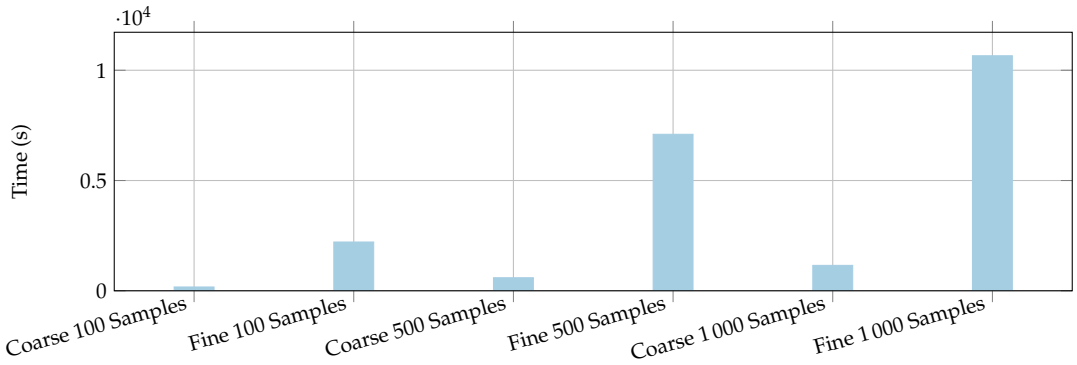


Figure 5.24. – Average runtimes for event arrival function generation and timing analysis depending on the ILP model and the number of used samples.

around 11 times longer when using the refined model compared the coarse model with the same number of sample points. The required runtime increases also with the number of used samples, yet directly proportional to it. Across all evaluated systems, the required runtime increases with an average factor of 0.76 times the number of used sample points. A reason for this factor being below 1.0 is the fact that the actual number of sample points required to calculate may be less than the defined one, as the last sample point is calculated first to know the maximum number of events. During sampling, each result is compared to this maximum value. In case the maximum has been reached, no more samples are to be taken, as the maximum has been already reached, lowering the effectively required number of sample points taken.

5.6.7 Conclusion

Taking into account the results of the previous evaluation aspects, a few general conclusions can be drawn for the generation of approximated event arrival functions with compositional timing analyses: In case of a multi-core system with a fixed priority-based bus arbitration, increasing the number of sample points while using the coarse-grained ILP model can in many cases greatly improve the precision with only a minor increase in required runtime. Additionally, the less cores in a system are present, the less number of sample points are required. If the highest analysis precision is required and a higher analysis runtime is not critical, a combination of a higher number of sample points (≥ 500) per event arrival function in combination with the refined ILP model is suggested. For multi-core architectures with a round robin-based bus arbitration, the most simple ILP model together with only a few sample points per event arrival function is adequate for the vast majority of cases for the use in a compositional timing analysis.

6 Code-Inherent Traffic Shaping

The results of the bus-aware instruction allocation presented in Chapter 4 demonstrated that including bus-induced timing effects into an optimization model for hard real-time systems can lead to significantly better results. Yet, the required optimization time of the presented approach drastically increases with more complex applications running on the cores due to the elaborate ILP model. Each shared memory access is modeled by its potential TDMA offset to derive the maximum number of cycles to be stalled. This very detailed level of modeling bus effects creates similar advantages and downsides as the very detailed joint worst-case execution time analysis for multi-core systems as described in Section 2.2.1. As the optimization model describes the bus-effects in a great detail, very small effects can be captured and may lead to a better worst-case timing behavior. This potentially enables a maximum gain of the optimization.

While a very detailed optimization model may achieve the best possible outcome, it often prevents scalability or the simple adaption to other scenarios. In case of multi-core-aware WCET-directed optimizations, this is amplified by the fact that the efficiency of an optimization is capped by the level of the analysis. For example, the previously presented bus-aware instruction allocation would most likely yield no gains if the WCET analysis tool were unable to analyze shared memory accesses down to a TDMA offset level. Hence, a very fine-grained optimization not only comes at the cost of a complex optimization problem to solve, but also with the requirement of a detailed analysis.

With the increasing complexity of modern multi-core architectures, a completely joint worst-case execution time analysis among all components seems infeasible. For example, even for considerably small programs (>100 source code statements) on a quad-core system to be analyzed, the WCET analyzer integrated into the WCC easily takes more than 2 hours to analyze for work-conserving bus arbitration strategies (e.g., round robin or fixed priorities). While the individual cores do not influence each other when using a TDMA bus arbitration, the cores' access behaviors have an effect on each other when a work-conserving bus arbitration is used. This leads to the further increase in the analysis complexity, as the cores can no longer be analyzed separately. As most modern bus-based multi-core architectures feature a round robin-based bus arbitration or a slightly adapted variant (e.g., Infineon TriCore AURIX TC277 [ZCMV15], Cobham GR740 (LEON4-based) [Cob19], NXP MSC811 (StarCore SC140-based) [NXP08]), it is reasonable to choose a level of timing analysis which features at least moderate scalability also for such features. Since the level at which a multi-core optimization is performed is indirectly linked to the granularity of the timing analysis, the choice of the timing analysis granularity also influences the optimization model.

As described in Chapter 5, so-called event arrival functions can be used to describe the access behavior of a program to, e.g., a shared resource. This metric can then be further used to derive the worst-case response time of a system or task which includes

bus-related timings. By extracting an event arrival function from the code-level of a program, a link between the microarchitectural level and the abstract system level can be drawn. While this enables timing analyses to be placed in between the two extremes of a low-level joint WCET analysis and a system-level analysis, it also allows a multi-core-aware optimization to use and specifically modify these event arrival functions to lower a task's WCRT or to meet other requirements.

One possible optimization to be done on the basis of event arrival functions is *traffic shaping*. Traffic shaping is a well-known technique in networks to improve end-to-end delays, avoid buffer overflows, or in general, to derive certain guarantees on transmission delays or throughput. Generally, it is used to increase the so-called Quality of Service (QoS) in the networks they are used in. The overall idea of a traffic shaper is to *shape* the traffic of a network node such that it adheres to a user-defined *traffic profile*. In order to do so, traffic shapers are either implemented as own components or as parts of a server which receive a stream of packets and potentially delay them. By delaying certain packets, they can guarantee that the user-defined traffic profile is always met. This typically translates to less bursts and a defined upper bound on the packets per time interval in the outgoing stream. These effects in turn can reduce overall network contention or the buffer requirement in network nodes down the line.

Traffic shapers can also be used in networks to give guarantees on the outgoing stream of network nodes. As the outgoing stream of a network shaper is bounded by its traffic profile, a corresponding upper event arrival function $\eta^+(\Delta t)$ can be used to describe the maximum outgoing network stream. By applying means of network calculus [LBT01], these event arrival functions can be used to derive precise and safe end-to-end delays, as well as buffer requirements.

The concept of traffic shaping has recently also been adapted in the domain of (hard) real-time systems. In this case, not network streams, but certain characteristics of the system are "shaped". This involves, e.g., the scheduling of tasks where the activation frequency of certain tasks is shaped [HHC+15, KT11, PL13] or the accesses to a shared bus [DN12b, WMT06, RTioBKBK+09, KSEE16]. In case of shaping the access profile to a shared bus, similar benefits as in the network domain can be achieved, such as lower end-to-end delays or lowered buffer requirements. While the shaping of accesses to a shared on-chip bus is discussed in the literature [WMT06], it is typically not directly applicable for actual COTS, as this would require traffic shaping hardware components between the individual cores and the bus (which are not existing). The traffic shaping functionality could be integrated by an adapted operating system by which every access to the shared bus is handled in this case. While there are approaches in the academic domain to enforce a certain access profile by means of an extended operating system for real-time systems [ZCMV15], the author is not aware of any existing functionalities of real-time operating systems to do so at the time of writing. Furthermore, this approach would bind the architecture to use a specific operating system - given the fact it exists for the architecture.

For distributed systems, the integration of a traffic shaper is in general possible by, e.g., using a gateway. Yet, this may not be desirable in every case, as the gateway's performance must be high enough to also act as a traffic shaper, or the system does not include a gateway, which would mean additional costs in terms of energy, space and money.

In the following, the concept of *code-inherent* traffic shaping is introduced to bypass these previously mentioned obstacles. By transforming the behavior of the program itself, the traffic shaping functionality is integrated *into* the program so to speak. The upper event arrival function of a program is thereby modified such that it resembles the output of an added traffic shaping component afterwards. This enables the benefits of traffic shaping principles to be used for shared on-chip bus architectures in COTS, distributed systems and more without the need for any additional hardware.

The core idea of the approach to be presented in the following sections is based on modifying the upper event arrival function of a given program by delaying the execution of specific event generating instructions. This allows the integration of a traffic shaping behavior independent from an additional higher level supervising operating system or required hardware, as the program itself adheres to a user-defined traffic profile. Typically, an event is defined as a shared bus access in the following. Yet, the approach can also be adapted for shaping different system characteristics, e.g., the call frequency of a specific function. An example would be to shape the call frequency to a wireless transmission function in order to optimize battery life. As the approach only handles the shaping of abstract events, it can be applied to anything what can be represented as an event as presented in Chapter 5.

This approach was first presented at the *International Conference on Embedded Software (EMSOFT)* in New York 2019 [OSF19].

The following Section 6.1 discusses related work. Section 6.2 will give a brief background on the principle of traffic shaping and event arrival functions, as well as how a given event arrival function can be efficiently checked whether it follows a certain traffic shaping profile or not. The idea of code-inherent traffic shaping for hard real-time systems is presented in Section 6.3. Here, Section 6.3.1 introduces the basic concept of code-inherent traffic shaping and Section 6.3.2 how this can be done in a WCET-aware manner. Section 6.4 introduces two different approaches, how this concept can be applied automatically: A greedy heuristic in Section 6.4.1 and an evolutionary algorithm in Section 6.4.2. Both approaches are then evaluated in Section 6.5. For the sake of evaluation, three different shapers are implemented which are discussed in Section 6.5.1. To detail the effects on an event arrival function, the different shapers are evaluated with both presented algorithms for an exemplary program in Section 6.5.2. A broader evaluation with a larger set of systems and parameters then follows in Section 6.5.3.

6.1 Related Work

Most work in the field of traffic shaping is located in the domain of comparably complex networks. The technique of traffic shaping is applied here to, e.g., improve latencies or to guarantee specific characteristics of streams inside the network. One of the arguably most popular traffic shaping algorithms, the so-called *leaky bucket* algorithm was proposed by Turner in 1986 [Tur86]. The key feature of this algorithm is an adjustable maximum average rate of the outgoing stream, as well as its maximum burstiness. The leaky bucket algorithm was then adopted by the Internet Engineering Task Force (IETF) into their *Specification of Guaranteed Quality of Service* [SPG97] as a suggested fundamental tool to enforce a certain level of Quality of Service (QoS) in the internet. As this thesis focuses on the domain of hard real-time systems, the

discussion of related work concerning the use of traffic shaping is limited to the domain of real-time systems.

The effect and performance of using greedy traffic shapers in hard real-time systems was evaluated by Wandeler et al. [WMT06] for two different applications. As the first scenario, a multi-core system with a shared bus and traffic shapers placed between each core and the bus is presented. It is shown that a worst-case end-to-end delay can be reduced by up 40% in the analyzed system, if the traffic shapers are introduced into the system. As a second scenario, traffic shapers are placed at the inputs of a single-core system to circumvent potential buffer overflows, isolate malicious input behavior and to improve worst-case end-to-end delays. Though the authors present how traffic shapers can be highly beneficial when used in hard real-time systems, it is left open how these are actually implemented.

Hamann et al. [HJRE06] presented a framework for design space exploration of heterogeneous embedded systems, in which traffic shaping is one possible design choice to improve certain latencies. They showed that while the insertion of a traffic shaper may increase the delay in certain areas of the system, the broad rest of the system may profit from it when chosen adequately. Similar insights were achieved by Schliecker et al. [SHRE08]. In the presented design space exploration of a multi-core system, an introduced traffic shaper can significantly decrease a set of worst-case delays in the system. As a compromise, another worst-case delay is increased.

A hardware-based approach to implement the concept of traffic shaping in a multi-core system was shown by Zhou and Wentzlaff [ZW16]. Each core is equipped with a physical traffic shaper such that the outgoing traffic of each core is well-defined. The authors conclude that a significant performance increase can be achieved with the proposed architecture when compared to statically allocating a certain bandwidth to each core. While the authors went beyond simulation and actually created and evaluated their architecture on actual silicon, modern multi-core COTS do not offer traffic shaping units.

Davis and Navet [DN12b] presented an approach to reduce jitter in CAN-based networks by employing traffic shapers. Here, the traffic shaping algorithm is integrated into the gateways connecting different networks. When a CAN message is sent from one network to another, it is temporarily buffered inside the gateway connecting both networks before it is passed on. In case multiple messages are buffered in the gateway and then quickly released subsequently, the gateway generates a burst of sudden messages in the outgoing network. This may happen if, e.g., the outgoing network is busy for a certain period and the gateway cannot pass on the messages in a timely manner. This bursty and uneven behavior increases the jitter of the message arrivals and may also increase worst-case end-to-end latencies. To reduce this so-called *queuing jitter*, Davis and Navet propose a traffic shaping algorithm inside the gateway nodes. The evaluation shows that the proposed approach can significantly reduce the queuing jitter.

Kumar and Thiele [KT11] presented an approach to apply the concept of traffic shaping to the execution of tasks. The underlying idea is to reduce the peak temperature of the system by selectively pausing actively running tasks such that the system cools down in these idle times. This is implemented inside a scheduler which follows a traffic shaping policy, whereas the microcontroller's execution time is the unit to be shaped. The enforced traffic shaping policy is designed in a way such that each

task is guaranteed to still meet its deadline, despite being throttled. The evaluation showed a reduction of peak temperature by up to 8 K.

A way to improve battery performance of mobile systems by means of traffic shaping was presented by Chiasserini and Rao [CR01]. The authors introduce the circumstance that the battery life of a mobile system can be improved if it is not constantly discharged with an average current, but rather in certain periods with a larger current and recovery periods with low current in between. Since wireless transmission is one of the major energy consumers in modern mobile systems [REWSP19], the underlying idea is to delay wireless transmissions to a certain degree and release them in bursts. This is done by shaping the requests to the wireless interface. Unfortunately, it is not described, how or on which level the actual shaping may be implemented.

Rahmani et al. [RTioBKBK+09] investigated the effects of traffic shaping inside an Ethernet-based automotive network. Different traffic shaping algorithms are evaluated in terms of their influence on the QoS of multimedia streams inside the network. The authors conclude that buffer requirements of network nodes as well as delay of packages can be reduced successfully if traffic shaping is applied.

The selective manipulation of the distribution of specific actions inside a program is also found in the area of preventing side-channel attacks. Wu et al. [WGSW18] present an approach to mitigate instruction- and cache-timing side-channels of a given program by automatically modifying it. Some of the presented are methods are to consistently read all elements of an array instead of only reading a single entry, to always read all cache lines during a memory access or to pre-load cache-lines. Using these concepts, the authors ensure that the execution time of the program is independent from "secret data" inside the program. Similarly, Wang et al. [WSW19] presented an approach to prevent power side-channels by altering the register allocation and memory locations on a code-level of potentially sensitive variables.

Whereas the related work either relies on (custom) hardware-based solutions or does not even specify how the traffic shaping may be implemented, the approach presented in this thesis is independent from hardware- or OS-based features. The related work also shows a variety of potential fields of applications. Beside using the approach of code-inherent traffic shaping for improving worst-case latencies in a multi-core system, it could also be used to reduce peak chip temperatures, improve battery life or mitigate side-channels.

6.2 Background

In the following section, a brief introduction on the topic of traffic shapers is presented. Subsequently, two approaches to evaluate whether an event arrival function $\eta^+(\Delta t)$ complies with a given traffic profile function $\sigma(\Delta t)$ or not are presented.

6.2.1 Traffic Shapers

Given an input stream which is bounded by an upper event arrival function $\eta^+(\Delta t)$, a traffic shaper processes this input stream and creates an output stream which in turn is bounded by a traffic profile $\sigma(\Delta t)$ by potentially delaying certain elements of the input stream. A more strict class of traffic shapers are the so-called *greedy* traffic shapers. A greedy traffic shaper delays elements of the input stream if the traffic profile $\sigma(\Delta t)$

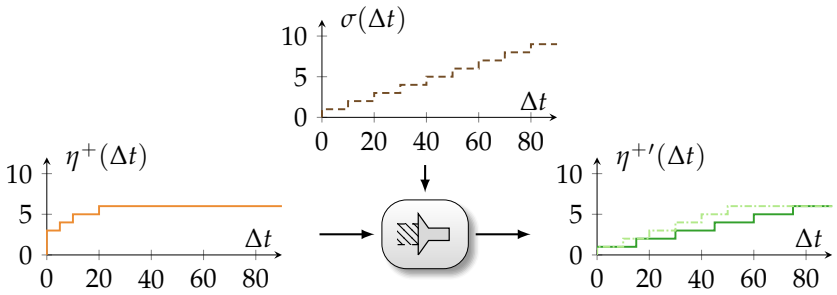


Figure 6.1. – The effect of a greedy traffic shaper (dash dotted line of $\eta^{+\prime}(\Delta t)$) and an exemplary lazy traffic shaper (solid line of $\eta^{+\prime}(\Delta t)$) on the resulting event arrival function.

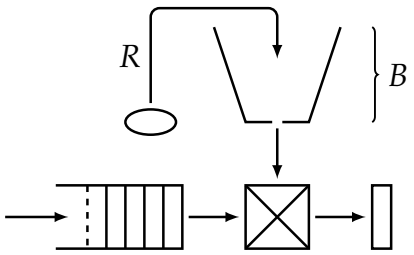


Figure 6.2. – Principle of token bucket traffic shaping.

would otherwise be violated, but passes them on *as soon as possible* [LBT01]. On the other hand, a so-called *lazy* traffic shaper also ensures that the outgoing event stream is bound by a traffic profile $\sigma(\Delta t)$, yet *does not* necessarily pass each event of the event stream as soon as possible [VBK16]. Figure 6.1 depicts the principle of greedy and lazy traffic shapers with the help of event arrival functions. $\eta^{+}(\Delta t)$ is the original upper event arrival function of an input stream and does not conform to any traffic profile. The profile function $\sigma(\Delta t)$ describes the maximum upper event arrival function to appear at the traffic shaper's output. The dash dotted line of $\eta^{+\prime}(\Delta t)$ shows the upper event arrival function of a greedy traffic shaper's output. In general, any traffic shaper enforces $\eta^{+\prime}(\Delta t) \leq \sigma(\Delta t)$ for every possible value of Δt. In this particular example, the traffic profile function $\sigma(\Delta t)$ realizes a simple rate control with at most 1 event every 10 time units. The original input stream clearly violates this traffic profile, as up to 5 events may occur in an interval of 10 time units. The traffic shaper enforces the traffic profile function $\sigma(\Delta t)$ and delays certain elements of the input stream such that the outgoing stream adheres to the traffic profile. Since a greedy traffic shaper only delays the stream as little as needed, the outgoing event arrival function $\eta^{+\prime}(\Delta t)$ of the greedy traffic shaper (the dash dotted line) is equal to $\min(\eta^{+}(\Delta t), \sigma(\Delta t))$.

The solid line of the outgoing event arrival function $\eta^{+\prime}(\Delta t)$ in Figure 6.1 shows the behavior of an exemplary lazy traffic shaper. The outgoing event arrival function $\eta^{+\prime}(\Delta t)$ is still bound by the given traffic profile function $\sigma(\Delta t)$, yet some events may be delayed longer than necessary. In this particular example, the lazy traffic shaper enforces a maximum of 1 event in a time interval with a length of 14 time units, although 2 events would be already permitted according to the traffic profile function $\sigma(\Delta t)$.

A widely used example of traffic shaping is the so-called *token bucket* algorithm, which is an equivalent formulation [Tan11] to the so-called *leaky bucket* algorithm proposed by Turner [Tur86] in 1986. It is still commonly used in many areas [KKXL17, QLI+17, UAA18]. Figure 6.2 shows an exemplary diagram of the algorithm, where the left-hand side queue represents the buffer for the input stream of the traffic shaper and the right-hand side the output stream. A so-called *token* is added to a collecting bucket at a defined timing rate R. A single token represents the right to send a specific amount of data. As depicted in Figure 6.2, the bucket does not have an unlimited volume, but can only hold at most B tokens. Newly arriving elements are stored in the traffic shaper's FIFO queue. If an element arrives at the traffic shaper's input and the bucket holds enough tokens to transmit this element, the element is directly passed through to the output and the corresponding number of tokens are removed from the bucket (as every transmission "costs" a certain amount of tokens). In case the bucket does not hold the required amount of tokens, the element is kept in the queue until the bucket has filled up with enough tokens and the element is passed through. Therefore, the maximum burstiness of the output stream is defined by the token bucket size B, whereas R can be seen as defining the rate of the outgoing stream. When setting the bucket size to a minimal value, the algorithm does not allow any bursts (as shown in the traffic profile $\sigma(\Delta t)$ in Figure 6.1). In the following, this specific case will be referred to as a "rate" shaping, as a maximum fixed rate without any bursts is enforced.

6.2.2 Traffic Profile Adherence Checking

As the core idea of a code-inherent traffic shaping is to transform a given program such that its upper event arrival function $\eta^+(\Delta t)$ adheres to a given traffic profile function $\sigma(\Delta t)$, a fundamental requirement is to check whether an event arrival function $\eta^+(\Delta t)$ *does* adhere to a profile function $\sigma(\Delta t)$ or not. Therefore, two different approaches to evaluate this are presented in the following. The first approach is applicable for any arbitrary traffic profile function $\sigma(\Delta t)$, whereas the second one can only be applied to a specific class of traffic profiles but requires significantly less effort.

It is assumed that the program's original upper event arrival function $\eta^+(\Delta t)$ is described using an ILP model as presented in Chapter 5. If not stated otherwise, a discrete time model with $t \in \mathbb{N}_0$ is assumed.

Arbitrary Traffic Profiles. A traffic shaper ensures that the upper bound of the outgoing traffic is bound by a given traffic profile which can be described by a corresponding traffic profile function $\sigma(\Delta t)$ in the time interval domain. If the traffic shaper's output stream is described using an event arrival function $\eta^+(\Delta t)$, $\eta^+(\Delta t)$ has to be lower than or equal to $\sigma(\Delta t)$ for every possible value of Δt. As both functions, $\eta^+(\Delta t)$ and $\sigma(\Delta t)$, are monotonically increasing and piecewise constant, it is sufficient to check at every discontinuity of $\eta^+(\Delta t)$ if the required relationship holds. This leads to a simple test as shown in Algorithm 6.1. Here, $\mathcal{K}$ is a set holding all required time interval lengths to check. This algorithm resembles the so-called *processor demand test* by Baruah [Bar03] which is used to evaluate the schedulability of a multi-task system. For each discontinuity, it is checked whether $\eta^+(\Delta t)$ is greater than $\sigma(\Delta t)$ or

Algorithm 6.1 Basic test for arbitrary profiles.

Inputs: $\eta^+(\Delta t)$ - Upper event arrival function to check for adherence; $\sigma(\Delta t)$ -Traffic profile function; $\mathcal{K}$ - Set of all time intervals to be checked
Output: A Boolean value which is true in case the event arrival function adheres to the traffic profile function and false if not.

```
1: for Δt in K do
2:     if η+(Δt) > σ(Δt) then
3:         return false
4:     end if
5: end for
6: return true
```

not. If this is true for any time interval, the algorithm return false, otherwise true. The complete upper event arrival function $\eta^+(\Delta t)$ of a given task can be derived using the Algorithm 5.2 presented in Section 5.4.2 (cf. Page 109). Yet, in case of a periodic system, $\eta^+(\Delta t)$ would have an infinite number of discontinuities. A sufficiently large time interval length up to which discontinuities have to be checked, would be the system's hyperperiod in this case. The system's hyperperiod is the least common multiple of the task's activation period and the traffic profile function's repetition period. Overall, this approach can quickly become infeasible for a larger number of points to test.

As an alternative approach, the traffic profile function $\sigma(\Delta t)$ can be directly described using additional constraints inside the ILP used to model the upper event arrival function. In this case, the original objective function (cf. Equation (5.13), Page 82) is replaced by the following one, which maximizes the difference between the two functions $\eta^+(\Delta t)$ and $\sigma(\Delta t)$:

$$\max : a^+_{\text{Total}} - \sigma\left(\sum_{i \in \mathcal{B}} z_i^+\right) \tag{6.1}$$

The traffic profile function $\sigma(\Delta t)$ is described in the ILP as well and the accumulated execution time is used as its corresponding parameter. Any previous constraints on the sub-paths's accumulated execution time (cf. Equation (5.12), Page 82) are removed. By solving the ILP with the updated objective function, a traffic profile adherence check is performed. In case the objective value is lower than or equal to zero, it is proven that $\sigma(\Delta t) \geq \eta^+(\Delta t)$ holds for all possible values of Δt. If the resulting objective value is greater than zero, the required condition ($\sigma(\Delta t) \geq \eta^+(\Delta t)$ for all possible Δt) is violated. Here, the objective value can be used as a figure of merit to estimate how much the event arrival function $\eta^+(\Delta t)$ violates the traffic profile function. Note that the ILP may also be unbounded in this case, indicating that the event arrival function violates the traffic profile function and has an overall greater gradient ($\lim_{\Delta t \to \infty} \eta^+(\Delta t) - \sigma(\Delta t) = \infty$).

This approach can be used for arbitrary traffic profiles, as long as the corresponding traffic profile function $\sigma(\Delta t)$ can be described using linear terms. In the following sections, examples for actual ILP descriptions of traffic profiles will be given. In the

rare circumstance that this is not possible, $\sigma(\Delta t)$ could be approximated by using linear terms.

Periodic Step Function Profiles. In case the traffic profile function follows the form of $\sigma(\Delta t) = Y \cdot \lceil \frac{\Delta t}{P} \rceil$, the effort for a profile adherence check can be further reduced. Here, the traffic profile function is a simple periodic step function with a step height of Y and a period of P.

Proposition 6.1. *If $\eta^+(P) \leq Y$ holds, then $\eta^+(\Delta t) \leq \sigma(\Delta t)$ holds for any $\Delta t \in \mathbb{N}_0$, given a profile function in the form of $\sigma(\Delta t) = Y \cdot \lceil \frac{\Delta t}{P} \rceil$.*

Proof. The value of $\eta^+(\Delta t)$ at every multiple period value is lower or equal to the profile function $\sigma(\Delta t)$:

$$n \in \mathbb{N}_0 \tag{6.2}$$

$$\eta^+(n \cdot P) \leq n \cdot \eta^+(P) \leq n \cdot Y \tag{6.3}$$

$$\sigma(n \cdot P) = Y \cdot \left\lceil \frac{n \cdot P}{P} \right\rceil = n \cdot Y \tag{6.4}$$

$$\Rightarrow \eta^+(n \cdot P) \leq \sigma(n \cdot P) \tag{6.5}$$

Equation (6.3) is exploiting the subadditivity of $\eta^+(\Delta t)$ and the required precondition given in Proposition 6.1. As $\sigma(\Delta t)$ only increases at points $\Delta t = n \cdot P + 1$ ($n \in \mathbb{N}_0$), and given the fact that $\eta^+(\Delta t)$ is monotonically increasing, the maximum deviation between the two curves can be found at points $\Delta t = n \cdot P$. As shown in Equation (6.5), the event arrival function is always below or equal to the shaper function for these points if the precondition is met. Hence, it is guaranteed that $\eta^+(\Delta t)$ is lower or equal to $\sigma(\Delta t)$ for any given Δt. □

This reduces the traffic profile adherence test to a single calculation of $\eta^+(\Delta t)$ for $\Delta t = P$. In case $\eta^+(P) > Y$, the resulting difference $\eta^+(P) - Y$ can be used as a figure of merit by how much the event arrival function violates the traffic profile.

Combined Adherence Check. Algorithm 6.2 combines the efficient check for arbitrary traffic profiles and simple periodic step functions and also returns more details if the traffic profile is violated. In case the given traffic profile function $\sigma(\Delta t)$ is a simple periodic step function as described in the previous paragraph, the simple adherence check is performed in Line 2. The "getEventsAndPath()"-function is described in Algorithm 6.3. This function solves the ILP description of the event arrival function and returns the objective value, as well as all basic blocks of the selected sub-path by the ILP solver as a set $\mathcal{A}_\eta$. To check, whether the event arrival function $\eta^+(\Delta t)$ violates the traffic profile function $\sigma(\Delta t)$ or not, $\sigma(P)$ is subtracted from the objective value in Line 3. Here, P is the period of the step function.

If the traffic profile function is not a simple periodic step function, the ILP description of the event arrival function is modified in Line 5. This includes the traffic profile function into the ILP description and sets the objective function to maximize the difference between the event arrival function and the traffic profile function. This updated ILP description is then also solved in Line 6. Due to the modified objective

Algorithm 6.2 combinedAdherenceCheck() – Efficient adherence check for arbitrary profiles with detailed results.

Inputs: $\eta^+(\Delta t)$ – ILP model of an upper event arrival function to check for adherence; $\sigma(\Delta t)$ – Traffic profile function; P – Period of $\sigma(\Delta t)$ if it is a simple periodic step function

Outputs: V – A figure of merit by how much the event arrival function violates the traffic profile function if $\eta^+(\Delta t)$ does not adhere to $\sigma(\Delta t)$, otherwise 0 is returned; $\mathcal{A}_\eta$ – A set which contains the basic blocks of a sub-path which leads to the event arrival function violating the traffic profile function if $\eta^+(\Delta t)$ does not adhere to $\sigma(\Delta t)$, otherwise an empty set is returned.

```
1:  if σ(Δt) is a simple periodic step function then
2:      (V, A_η ) ← getEventsAndPath( η⁺(P) )
3:      V ← V - σ(P)
4:  else
5:      Modify η⁺(Δt) according to Equation (6.1)
6:      (V, A_η ) ← getEventsAndPath( η⁺(Δt) )
7:  end if
8:  if V ≤ 0 then
9:      return (0, ∅)
10: else
11:     return (V,A_η)
12: end if
```

function, the resulting objective value already represents the maximum difference between $\eta^+(\Delta t)$ and $\sigma(\Delta t)$, hence no subtraction as in Line 3 is required.

If V is lower than or equal to 0, $(0, \emptyset)$ is returned in Line 9. This means that the event arrival function does not violate the traffic profile function. Otherwise, the tuple $(V, \mathcal{A}_\eta)$ is returned in Line 11. V represents a figure of merit, by how much the event arrival function violates the traffic profile. The set $\mathcal{A}_\eta$ contains all basic blocks of a sub-path which leads to a violation of the traffic profile.

Algorithm 6.3 details how the "getEventsAndPath()"-function used in Algorithm 6.2 is implemented. The algorithm solves a given ILP model describing an upper event arrival function and returns the objective value, as well as a set of all basic blocks which are part of the sub-path chosen by the ILP solver. Initially, the set $\mathcal{A}_\eta$ is an empty set in Line 1. Then the ILP model is solved in Line 2 and the resulting objective value is stored in the variable N. To determine which basic blocks are part of the sub-path chosen by the ILP solver, the for-loop over Lines 3 to 7 iterates over all basic block of the program. In case the execution count of a basic block i is larger than 0 (p_i refers to ILP variable describing the number of executions of a basic block i on the sub-path), the basic block is part of the sub-path and is added to the set $\mathcal{A}_\eta$ in Line 5. Finally, the tuple $(N, \mathcal{A}_\eta)$ is returned in Line 8.

6.3 Code-Inherent Traffic Shaping

In the following section, the idea and integration of *code-inherent* traffic shaping is introduced. As a prerequisite, a system with code allocated to private memories is

Algorithm 6.3 getEventsAndPath() – Solves the ILP model describing an upper event arrival function and returns the objective value, as well as the selected sub-path.

Inputs: $\eta^+(\Delta t)$ – ILP model of an upper event arrival function to check for adherence
Outputs: N – Objective value of the ILP after solving; $\mathcal{A}_\eta$ – A set containing all basic blocks which are part of the selected sub-path leading to the objective value.

1: $\mathcal{A}_\eta \leftarrow \emptyset$
2: $N \leftarrow$ solveILP($\eta^+(\Delta t)$)
3: **for** i in $\mathcal{B}$ **do**
4: **if** $p_i > 0$ **then**
5: $\mathcal{A}_\eta = \mathcal{A}_\eta \cup \{i\}$
6: **end if**
7: **end for**
8: return (N, $\mathcal{A}_\eta$)

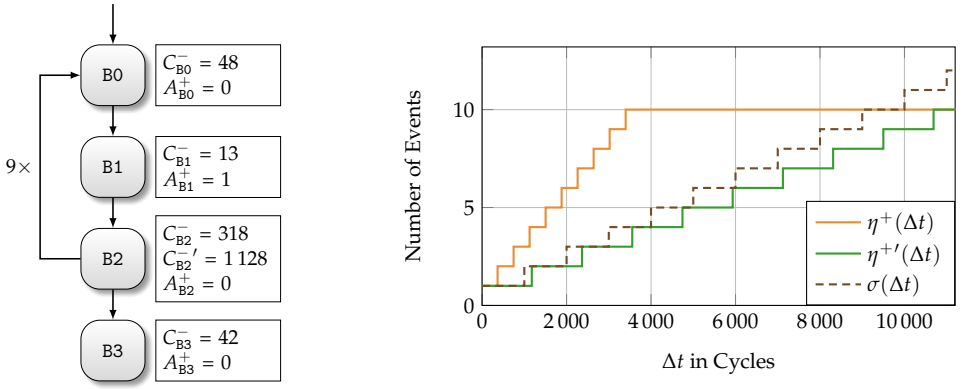
assumed, such that the insertion of additional delaying instructions does not cause additional bus accesses. To increase precision and efficiency of the code-inherent traffic shaping, it is assumed that a (sub) basic block contains at most 1 event. This does not mean any loss of generality, as the principle of sub basic block splitting (presented in Section 5.3.7, Page 104) can be applied beforehand to the program.

6.3.1 Basic Principle

The previous Chapter 5 presented how an accurate and safe event arrival function can be extracted on assembly code level, given the program's control-flow graph, the limits on the execution time per basic block and the number of events generated per block. Given an upper event arrival function of a program, it can be determined whether this event arrival function (and thus the program's behavior) conforms with a desired traffic profile or not. As the derived event arrival function is safely derived from the code-level and covers *all* possible paths, this compliance check is able to *guarantee* that the program's behavior adheres to the traffic profile at any time.

As an illustration, an exemplary control-flow graph is shown in Figure 6.3(a). Beside each basic block `B`, its best-case execution time $C_{\mathtt{B}}^-$ and its maximum number of generated events $A_{\mathtt{B}}^+$ are depicted. Figure 6.3(b) shows the corresponding upper event arrival function $\eta^+(\Delta t)$ of the program, as well as two additional curves, which are discussed in the upcoming paragraphs. As only basic block `B1` generates an event and is executed 10 times (note the loop is tail-controlled, hence the back-edge is executed one time less than the loop body), the event arrival function is a simple step function which converges at 10 events. An event can be, e.g., the access to a shared memory or the sending of a message over a bus via a memory-mapped register.

As a traffic profile function $\sigma(\Delta t)$, a simple step function is chosen here as an example shown in Figure 6.3(b). Here, at most 1 event is allowed every 1 000 cycles which corresponds to a very simple rate control which does not allow any bursts. The given example program clearly does not adhere to this traffic profile, as it generates up to 3 events in a time interval of 1 000 cycles. In fact, the upper event arrival function $\eta^+(\Delta t)$ already surpasses the traffic profile function at $\Delta t \approx 370$ cycles.



(a) An exemplary control-flow graph with annotated information.

(b) A traffic profile function and the corresponding event arrival functions.

Figure 6.3. – An exemplary control-flow graph (a) and its corresponding upper event arrival functions before ($\eta^+(\Delta t)$) and after modification ($\eta^{+\prime}(\Delta t)$), as well as the desired traffic profile function ($\sigma(\Delta t)$) (b).

In order to shape the upper event arrival function of the program such that it will comply to the traffic profile, it can be modified on a code-level. In this particular case, the program has to be modified such that the minimum execution time between 2 events is at least 1 000 cycles. This can be achieved by, e.g., increasing the execution time of basic block `B2`, since it has to be executed between the generation of any two subsequent events. Therefore, the minimum execution time of basic block `B2` is increased from 318 cycles to 1 128 cycles by inserting delaying instructions. Instead of basic block `B2`, also basic block `B0` or `B1` (or all of them) could be chosen, `B2` is only used here as an example. This is denoted in Figure 6.3(a) as the modified execution time $C^-_{B2}{}'$. A delaying instruction does not change the state of the program and can be, e.g., a simple `NOP` instruction. The updated upper event arrival function with the inserted delaying instructions is depicted in Figure 6.3(b) as $\eta^{+\prime}(\Delta t)$. As can be seen, the modified program's behavior now adheres to the traffic profile, as its upper event arrival function $\eta^{+\prime}(\Delta t)$ is lower than or equal to the traffic profile function $\sigma(\Delta t)$ for any value of Δt. This principle can be seen as a *code-inherent* traffic shaping, as the behavior of the modified program is now bounded by the profile function $\sigma(\Delta t)$, similar as if a traffic shaper would have processed its outputs.

While the modified event arrival function $\eta^{+\prime}(\Delta t)$ complies with the desired traffic profile, the applied shaping does not correspond to a *greedy* shaper, as this would require that $\eta^{+\prime}(\Delta t)$ is equal to $\sigma(\Delta t)$ up to $\Delta t = 10\,000$ cycles in this case (as a greedy shaper emits a request as soon as possible). There are two reasons why an ideal greedy shaping cannot be implemented using code-inherent shaping:

1. The timing granularity of the added delay is limited by the underlying architecture. As the minimal delay to be added is realized by a single delaying instruction, the minimal time required to fetch and execute an instruction limits the finest granularity achievable.

2. Depending on the structure of the control-flow graph, the insertion of delaying instructions may inevitably shift the whole event arrival function, and not only

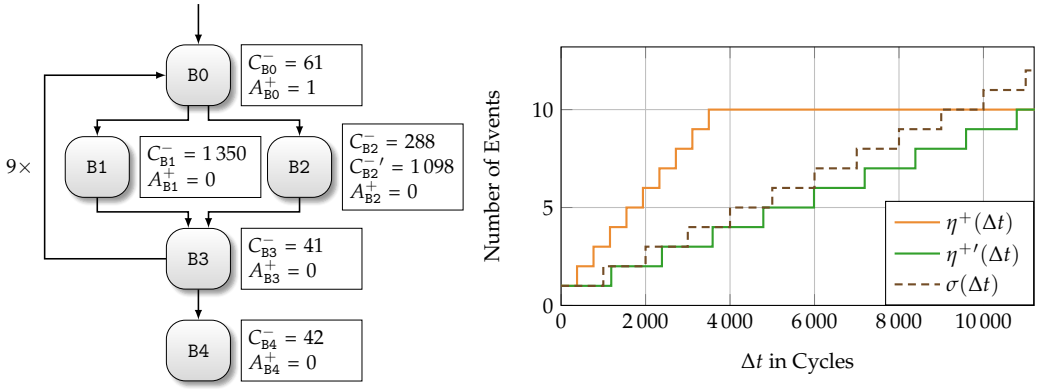
a local event. In these cases, it is impossible to perform an ideal *greedy* code-inherent traffic shaping without additional measures like, e.g., rescheduling of code.

The addition of functionally not required instructions into a program may raise questions concerning the validity of this in the domain of safety-critical real-time systems. A commonly known and used set of development guidelines and rules for software in critical systems are the MISRA C guidelines [MIR13]. According to them, a program is not allowed to have any *dead code*, which is defined as "*any operation that is executed but whose removal would not affect program behaviour*". At a first glance, this would forbid the application of code-inherent traffic shaping as just presented in critical systems. Yet, the guidelines clarify that "*the behavior of an embedded system is often determined not just by the nature of its actions, but also by the time at which they occur*". Although the additional delaying instructions do not change the functional behaviour of a program, they may significantly change the timing when specific actions occur in the program on purpose. Therefore, the additional delaying instructions are not classified as dead code, as they do affect the program's behavior. Generally speaking, any insertion of code into a safety-critical system by a compiler must comply with a corresponding functional requirement. And if the adherence to some timing profile and schedulability of tasks are part of the requirements, code-inherent traffic shaping is feasible for critial systems.

6.3.2 WCET-Aware Code-Inherent Traffic Shaping

While in the previously presented example, the program's access behavior could be shaped to be compliant with a given traffic profile using code-inherent shaping, it obviously increased the program's worst-case execution time. Assuming that for the previous example the basic blocks' WCETs are equal to the BCETs, the program's worst-case execution time increased from 3 832 cycles to 11 932 cycles. This inevitable increase in WCET stems from the program's very simple structure (only a single execution path exists). In this section, it is shown that ideally, the concept of code-inherent traffic shaping can be applied in a WCET-aware way by exploiting a program's structure.

Figure 6.4(a) shows another exemplary control-flow graph. In contrast to the previous control-flow graph, Figure 6.4(a) has an if-then-else-structure inside the loop, whereas only the loop entry `B0` is generating an event. The corresponding upper event arrival function $\eta^+(\Delta t)$ and the desired traffic profile function $\sigma(\Delta t)$ are shown in Figure 6.4(b). As there is still only one event generated inside the loop and the loop's upper bound did not change, the upper event arrival function resembles the previous one closely. Again, the program's event arrival function does not adhere to the desired traffic profile. In order to shape the event arrival function, additional delaying instructions are added to basic block `B2`, increasing the previous timing $C^-_{\texttt{B2}}$ from 288 cycles to 1 098 cycles. With this modification, the program's upper event arrival function now conforms with the traffic profile function $\sigma(\Delta t)$ as can be seen in Figure 6.4(b). Yet, the additional delaying instructions in `B2` do not change WCET of the program, which is 14 562 cycles before and after the modification. This is due to the program's worst-case execution path and the choice in which basic block the delaying instructions should be placed. The unmodified program's WCEP contains



(a) An exemplary CFG with annotated information including an if-else-statement. **(b)** A traffic profile function and the corresponding event arrival functions.

Figure 6.4. – A second exemplary control-flow graph (a) and its corresponding upper event arrival functions before ($\eta^+(\Delta t)$) and after modification ($\eta^{+\prime}(\Delta t)$), as well as the desired traffic profile function ($\sigma(\Delta t)$) (b).

the basic blocks `B0`, `B1`, `B3` and `B4`. Yet, the path along which the maximum number of events may be generated in the smallest time interval includes block `B2` and not `B1` (as `B2` has a lower BCET). As the upper event arrival function describes the *maximum* number of events in a certain time interval, it is defined by the *minimum* time between two or more events. Therefore, only the program's BCET is increased, while the event arrival function is successfully shaped. This enables the possibility of a WCET-aware code-inherent shaping which aims at only modifying parts which are not part of the WCEP (and not cause a switch of it).

6.4 Integration of WCET-Aware Traffic Shaping Behavior

While the previous section introduced the concept of a WCET-aware code-inherent traffic shaping, the actual method *how* it is done is left open. This can be reduced to the questions where and how many delaying instructions need to be inserted such that the event arrival function adheres to the traffic profile function, while the WCET is increased as little as possible. More formally put, the problem to be solved is as follows:

Given a program `P` and a profile function $\sigma(\Delta t)$, determine the locations (i.e., which basic blocks) and amount of delaying instructions to be inserted into the program `P`, such that the resulting event arrival function $\eta^+(\Delta t)$ will be less than or equal to $\sigma(\Delta t)$ for all Δt while increasing the WCET as little as possible.

Two approaches are presented in the following to solve this problem. The first is a greedy heuristic, whereas the second one is an evolutionary approach. The greedy heuristic is chosen as a scalable approach with an adjustable level of precision, while the evolutionary approach is chosen due the versatility and wide-spread application of evolutionary algorithms.

6.4.1 Greedy Heuristic

The overall idea of the greedy heuristic is to identify basic blocks which are part of a sub-path which causes the traffic profile to be violated but *not* part of program's current worst-case execution path. Referring to the previous exemplary control-flow graph and traffic profile in Figure 6.4, this attribute only applies to basic block `B2`. If such a basic block can be found, delaying instructions are added until either the traffic profile is met or the WCEP switches and includes the chosen basic block. In the latter case, a new basic block is searched. Algorithm 6.4 lists the complete algorithm.

The heuristic maintains a map $\mathbf{N}$ with basic blocks as keys to keep track of the added delay per basic block. $\mathbf{N}$ is initialized with 0 for all basic blocks of the program in Line 4. The main part of the algorithm starts in Line 7 where a profile adherence check is performed at the beginning of every main iteration. The profile adherence check is carried out as described in Algorithm 6.2. If the traffic profile is not violated, the returned value V is 0, otherwise V is a positive integer representing a figure of merit by how much the traffic profile is violated. $\mathcal{A}_\eta$ is a set containing all basic blocks of a sub-path violating the traffic profile. Referring to the previous exemplary control-flow graph and traffic profile in Figure 6.4, $\mathcal{A}_\eta$ could be $\{\texttt{B0}, \texttt{B2}, \texttt{B3}\}$ here. In case there are multiple paths leading to a violation of the traffic profile, an arbitrary sub-path is chosen. This set of basic blocks can be retrieved easily from the ILP model required to be solved for the profile adherence check as shown in Algorithm 6.3. If the program's behavior conforms with the traffic profile, $\mathcal{A}_\eta$ is an empty set (and V equals 0). In this case, the algorithm terminates in Line 9 by breaking the outermost loop. Otherwise, a WCET analysis is carried out in Line 11 which returns the program's WCET C^+, as well as the set of all basic blocks which are part of the worst-case execution path $\mathcal{A}_\mathrm{W}$. To speed up the algorithm, this WCET analysis, as well as the other WCET analyses in this algorithm, is simplified. In particular, the initial steps, such as the microarchitectural analysis (cf. Section 2.2.1, Page 15) to receive the WCET of each (unmodified) basic block, are only performed once. The additional delay $\mathbf{N}[i]$ of a basic block i is included in the final path analysis step by simply increasing the WCET value of this basic block by $\mathbf{N}[i]$.

Subsequently, a subset of $\mathcal{A}_\eta$ is created by removing all basic blocks from the set which are also part of the WCEP. This modified set is named $\mathcal{A}_\eta{}'$ in Line 12. $\mathcal{A}_\eta{}'$ contains all basic blocks of a sub-path violating the traffic profile but not being part of the WCEP. In case this set is not empty, an arbitrary basic block `B` from the set is chosen and a binary indicator variable Z is set to false in Lines 13 to 15. Z indicates that the chosen basic block `B` is not part of the WCEP. This information is used later in order to select a different basic block in case of a WCEP switch.

In case the modified set $\mathcal{A}_\eta{}'$ is empty, all basic blocks of the current sub-path violating the traffic profile are also part of the WCEP. This would happen, e.g., in the case depicted in Figure 6.3. As all basic blocks of the current traffic profile violating sub-path are also on the program's WCEP, the program's WCET will inevitably grow due to shaping, as additional delay has to be inserted to at least one of these basic blocks in order to meet the traffic profile's requirement. To mitigate the side-effects on the WCET, the basic block with lowest worst-case execution count is chosen from the set $\mathcal{A}_\eta$ in Line 17. In case there are multiple basic blocks in $\mathcal{A}_\eta$ sharing the same minimal worst-case execution count (WCEC), an arbitrary block among those is chosen. The reasoning is here that an additional delay in a basic block with a very

Algorithm 6.4 Greedy heuristic for WCET-aware traffic shaper integration.

Inputs: ϵ – Relative threshold for searching the minimum number of delaying instructions to be inserted, $\epsilon \in [0,1]$; μ – Lowest number of cycles of delay by using a single delaying instruction; $\eta^+(\Delta t)$ – ILP model of an upper event arrival function to check for adherence; $\sigma(\Delta t)$ – Traffic profile function; P – Period of $\sigma(\Delta t)$ if it is a simple periodic step function
Outputs: $\mathbf{N}$ – Map containing the additional delay for each basic block
for i in $\mathcal{B}$ **do**
    $\mathbf{N}[i] \leftarrow 0$
end for
while true **do**
    $(V, \mathcal{A}_\eta) \leftarrow$ combinedAdherenceCheck$(\eta^+(\Delta t), \sigma(\Delta t), P)$
    if $V = 0$ **then**
        break
    end if
    $(C^+, \mathcal{A}_W) \leftarrow$ calcWCET$(\mathbf{N})$
    $\mathcal{A}_\eta{}' \leftarrow \mathcal{A}_\eta \backslash \mathcal{A}_W$
    if $\mathcal{A}_\eta{}' \neq \emptyset$ **then**
        $\mathtt{B} \in \mathcal{A}_\eta{}'$
        $Z \leftarrow$ false
    else
        $\mathtt{B} \in \{\mathtt{x} | \mathtt{x} \in \mathcal{A}_\eta, \mathrm{WCEC}(\mathtt{x}) \leq \min_{\forall \mathtt{i} \in \mathcal{A}_\eta} (\mathrm{WCEC}(\mathtt{i}))\}$
        $Z \leftarrow$ true
    end if
    do
        $L \leftarrow \mathbf{N}[\mathtt{B}]$
        $\mathbf{N}[\mathtt{B}] \leftarrow (\mathbf{N}[\mathtt{B}] = 0 \; ? \; \mu : \mathbf{N}[\mathtt{B}] \cdot 2)$
        $(C^{+\prime}, \mathcal{A}_W) \leftarrow$ calcWCET$(\mathbf{N})$
        $(V, \mathcal{A}_\eta) \leftarrow$ combinedAdherenceCheck$(\eta^+(\Delta t), \sigma(\Delta t), P)$
        if (Z = false) and $(C^{+\prime} > C^+)$ **then**
            break
        end if
    while $(V > 0)$ and $(\{\mathtt{B}\} \cap \mathcal{A}_\eta \neq \emptyset)$
    $U \leftarrow \mathbf{N}[\mathtt{B}]$
    $\mathbf{N}[\mathtt{B}] \leftarrow (L + U)/2$
    while $((U - L)/L > \epsilon)$ and $(\mathbf{N}[\mathtt{B}] \geq \mu)$ **do**
        $(C^{+\prime}, \mathcal{A}_W) \leftarrow$ calcWCET$(\mathbf{N})$
        $(V, \mathcal{A}_\eta) \leftarrow$ combinedAdherenceCheck$(\eta^+(\Delta t), \sigma(\Delta t), P)$
        if $(V = 0)$ or $(\{\mathtt{B}\} \cap \mathcal{A}_\eta = \emptyset)$ or $(\overline{Z}$ and $(C^{+\prime} > C^+))$ **then**
            $U \leftarrow \mathbf{N}[\mathtt{B}]$
        else
            $L \leftarrow \mathbf{N}[\mathtt{B}]$
        end if
        $\mathbf{N}[\mathtt{B}] \leftarrow (L + U)/2$
    end while
    $\mathbf{N}[\mathtt{B}] \leftarrow U$
end while
return $\mathbf{N}$

high worst-case execution count will have a large impact on the WCET increase, as this delay will be multiplied by the block's WCEC. To indicate that the chosen basic block B is on the program's worst-case execution path, Z is set to true in Line 18.

After the basic block B is chosen where to insert the additional delay, the loop spanning from Line 20 to 28 keeps increasing the delay. At the start of each iteration of the do-while-loop, the current amount of delay added to the basic block B is stored to the variable L in Line 21. Subsequently, the delay is increased. In case no delay was yet added to the basic block B, a minimal delay of μ is added. μ is architecture-dependent and represents the minimal delay that can be added to a basic block. If delay was already added to basic block B previously, the amount of delay is doubled in Line 22. The idea here is to quickly find an amount of delay, which will cause the sub-path including the basic block B to not violate the traffic profile anymore. This delay will be refined later. Subsequently, the program's WCET is analyzed and a profile adherence check is performed in Lines 23 and 24. In case the program's WCET grew due to the increased delay in basic block B and B was not part of the WCEP previously, the do-while-loop is exited in Line 26. This occurs when the added delay in basic block B becomes large enough to cause a switch of the worst-case execution path, now including B. In this case, further increasing the delay of basic block B is stopped, as there are potentially other basic blocks on the traffic profile violating sub-path which are not yet part of WCEP. Otherwise, it is checked if the traffic profile is still violated and basic block B is still part of the violating sub-path. If at least one of the two conditions is not met, the do-while-loop is exited (Line 28), as there is no need the delay of B to be increased currently anymore.

At Line 29, the current amount of delay added to basic block B is stored to the variable U. At this point, enough delay was added to basic block B to cause a change. This change is either the increase of the program's WCET in case B was not part of the WCEP before (Z is false), or that B is no longer part of the current traffic profile violating sub-path. Either way, the current amount of delay added to basic block B $\mathbf{N}[\mathtt{B}]$ may be significantly larger than required to lead to this change, as the additional amount of delay is doubled in each iteration of the previous do-while-loop. As the overall aim is to meet the traffic profile while being as least invasive as possible, the following loop from Line 31 to 40 tries to reduce the amount of delay again. The minimal amount of delay required to trigger this change is to be found in the range of $[L, U]$, where L is the lower bound on the delay and U is the upper bound. Therefore, a binary search is carried out to find this minimal amount of delay. The loop is executed as long as the added delay is greater than or equal to the minimal delay possible μ and the distance between the upper and lower bounds (normalized to the lower bound) is greater than a user-given threshold ϵ. The threshold ϵ is a positive real constant to control a trade-off between accuracy and the algorithm's runtime. If set to 0, the binary search will be carried out until the exact amount of delay to trigger the change has been found ($U = L$). The greater ϵ is chosen, the quicker the binary search terminates, as the allowed difference between the lower and upper bounds grows.

At the start of each iteration of the binary search, the WCET is analyzed and a traffic profile adherence check is performed with the current added delay to basic block B (Lines 32 and 33). If this amount of delay is enough to trigger the change (Line 34),

this delay is stored as the upper bound. This can be caused by one of the following conditions:

1. The program does not violate the traffic profile at all anymore ($V = 0$).
2. The current basic block is not part of the sub-path returned by the adherence check anymore ($\{\mathtt{B}\} \cap \mathcal{A}_\eta = \emptyset$).
3. The current basic block was not part of the WCEP previously but is now ($\overline{Z}$ and $(C^{+\prime} > C^{+})$).

Otherwise, the lower bound is updated. Afterwards, the added delay of basic block `B` is changed to the midpoint of L and U in Line 39. Finally, when the binary search terminates, the last found upper bound U is chosen as the amount of delay to be added to basic block `B`.

After performing Algorithm 6.4, the basic blocks are adapted to actually include the delay. This is done by inserting a minimal loop at the beginning of each basic block which has a delay. The loop bound of each loop has to be set according to the determined delaying cycles and the number of cycles required for one loop iteration. The exact structure of such a loop and the number of cycles required for one loop iteration is architecture-dependent and has to be determined beforehand. While it would be possible to simply insert a sequence of `NOP` operations instead of using loops, this would be highly inefficient in terms code size.

6.4.2 Evolutionary Algorithm

The previously discussed greedy heuristic to find the locations and amount of delay to be added to the basic blocks to shape the program's upper event arrival function while trying to keep the WCET increase as low as possible may not find the ideal basic blocks in case blocks on the WCEP have to be modified. While the choice of a basic block with the lowest WCEC is an adequate guess, it might not lead to the best solution. As an alternative, an approach based on evolutionary algorithms is presented in the following. The evolutionary algorithm is bi-criterial with WCET and the traffic profile violation being the guiding objectives. In the following, the characteristics of the evolutionary algorithm are discussed.

Gene Composition. Each individual of a population is represented as a list of unsigned integers. Each integer represents the amount of delay to be added to a corresponding basic block of the program. Therefore, each individual's genome consists of $|\mathcal{B}|$ unsigned integers, which is the total number of basic blocks in the program.

Initial Population. Given an initial population with N_P individuals (where $N_\mathrm{P} \geq 4$), two individuals are created to represent corner cases. The first corner case is an individual with no delay added to any basic block, resulting in the lowest possible WCET increase (namely 0). The second corner case is a very heavily modified individual. To *each basic block* of this individual, a significant amount of delay is added. This individual will most likely result in a very low traffic profile violation, possibly even none (yet with a large WCET increase). The remaining individuals of the initial population are split in two groups: A guided and an unguided group. For the guided group, $(N_\mathrm{P} - 2)/2$ individuals are created. A random amount of delay is drawn (uniformly distributed) from the range $[0, M_\mathrm{GA}]$ for every basic block which is

part of the traffic profile violating sub-path according to an initial profile adherence check for each individual in this group. M_{GA} is a user-defined constant denoting the maximum amount of delay to be added to a basic block in the initial population. This should guide this group of individuals to good solutions at the beginning of the evolutionary algorithm. The remaining individuals are initialized in an unguided manner. Random delays in the range of $[0, M_{\mathrm{GA}}]$ are drawn and are added for *all* basic blocks in each individual for this group. The individuals of the guided group are supposed to increase chances of good starting points in the initial population, whereas the individuals of the unguided group increase the diversity. Additionally, the first two corner cases individually will most likely survive through many generations, as they will yield very good fitness in terms of WCET increase or traffic profile violation, and hence form promising recombination partners.

Fitness Function. The two criteria evaluated in the fitness function are the WCET of an individual and the traffic profile violation. The WCET of an individual is determined by performing a WCET analysis. Similar to the greedy heuristic presented in the previous section, the WCET analysis used in the fitness function is simplified to speed up the algorithm, as nothing beside the worst-case timings of the individual basic blocks change. Here, only the path analysis with the updated worst-case timings of the basic blocks including the delay is performed. The traffic profile violation is represented by the figure of merit resulting from a profile adherence check (see Section 6.2.2), describing by how much an individual is still violating the desired traffic profile. The actual fitness value of an individual is determined using the multiobjective fitness assignment of the SPEA2 algorithm [ZLT01], where the fitness of an individual i corresponds to the number of individuals dominating the individual i. The SPEA2 algorithm maintains two sets of individuals, one containing the active individuals of the current generation and one filled with the best individuals found so far, the so-called *archive*. For determining the fitness of an individual i, first the number of individuals that are dominated by i (considering both sets of individuals: the current population, as well as the archive) is determined. Subsequently, a so-called *raw fitness* of each individual i is calculated by summing up the number of individuals *dominated* of each individual *dominating* the individual i. This has the intention to increase the strength of the domination relationship, the more overall individuals are dominated by one individual. The lower the raw fitness value is, the fitter the individual. Hence, the SPEA2 algorithm always tries to minimize the fitness. Furthermore, a so-called density is added to to the raw fitness value to set the final fitness value of an individual. This helps further discriminate between individuals with identical raw fitness values (i.e., individuals which do not dominate each other). The density of an individual i is depending on the proximity (in the objective space) of the other individuals with an identical raw fitness value. The less individuals are in close proximity, the lower the additional density value. This has the intention to give individuals in a less "crowded" area of the objective space an advantage in order to keep this pareto-point existing. Note that the fitness values of both sets of individuals, the active population and the archive, are jointly determined at each generation.

The fitness value of each genome is cached internally such that, if an identical genome re-appears in the population in a later generation, the fitness value does not need to be re-calculated.

Mutation. The mutation used in the evolutionary algorithm is not completely random but guided in order to help the algorithm to converge faster. In case the profile adherence check of an individual returns a violation, the mutation process is only allowed to *add* delay to basic blocks of this individual which are part of the violating sub-path. Similarly, if the profile adherence check of an individual results no violation, the mutation can only *remove* previously added delay from basic blocks.

Each basic block eligible for mutation (as described in the previous paragraph) of an individual is mutated with a user-defined mutation probability P_{M}. In case a basic block is mutated and no delay was added yet to the block, a delay of M_{GA} is added if the individual is violating the traffic profile. If the individual does not violate the traffic profile, the mutation does not alter the block in this case, as the basic block does not have any delay to be reduced. If a basic block is mutated and already has delay added, the delay is increased by a factor randomly drawn (uniformly distributed) from the range $[1.0, 2.0]$ if the individual is violating the traffic profile. Otherwise, the delay of this basic block is reduced by a factor randomly drawn from the range $[0.0, 1.0]$.

Crossover. A single point crossover between two individuals is performed. The crossover point is randomly determined (uniformly distributed) from the range $[0, |\mathcal{B}| - 1]$. The offspring's genome is created by adopting the first individual's genome until the crossover point and the second individual's genome for the second part.

6.5 Evaluation

In the following section, the two presented approaches for performing a WCET-aware code-inherent traffic shaping are evaluated and their effects are discussed. At first, different shaping algorithms are discussed and how they can be described for an efficient profile adherence check. These shaping algorithms are then used to evaluate the greedy heuristic and the evolutionary algorithm.

6.5.1 Implemented Shapers

For the evaluation, three different shaping algorithms were implemented. The overall approach is not limited to these 3 presented shapers, but can be applied to any shaping algorithm which can be described using a corresponding traffic profile function $\sigma(\Delta t)$.

Lazy Token Bucket. The *lazy token bucket* (LTB) shaper has the traffic profile function $\sigma(\Delta t)$ of a regular token bucket shaper with a bucket size of B and a refill rate of R. The prefix *lazy* is added, as the principle of code-inherent traffic shaping cannot guarantee a *greedy* traffic shaping behavior as discussed in Section 6.3.1, whereas a regular token bucket shaper is typically considered as greedy. It is assumed that the processing of a single event costs a single token. Yet, this is no restrictive assumption,

as any other cost per event can be assumed by simply multiplying the number of events with an associated cost. The corresponding traffic profile function is as follows:

$$\sigma(\Delta t) = \begin{cases} 0 & \text{if } \Delta t \leq 0, \\ B + \lceil R \cdot \Delta t \rceil & \text{else.} \end{cases} \tag{6.6}$$

With $B \in \mathbb{N}_0$ and $R \in \mathbb{R}^+, R > 0$. The token bucket shaper allows a bursty behavior, as it has an initial step of height $B + \lceil R \rceil$, which allows up to $B + \lceil R \rceil$ events in a single time unit to be generated. The refill rate R enforces a minimum distance between these bursts, as the bucket is only refilled with this rate. The lazy token bucket shaper's profile adherence check is implemented by describing $\sigma(\Delta t)$ inside the upper event arrival function's ILP model. This is done using the following constraints:

$$\sigma \geq B + R \cdot \sum_{i \in \mathcal{B}} z_i^+ \tag{6.7}$$

$$\max : a_{\text{Total}}^+ - \sigma \tag{6.8}$$

σ is represented as an integer variable inside the ILP model. As the objective is set to maximize the difference between the number of events generated along a sub-path and the traffic profile $\sigma(\Delta t)$, where Δt is the execution time for this sub-path, only the lower bound of σ needs to bounded inside the ILP model.

Lazy Rate. The *lazy rate* (LR) shaper is a simplified version of the previous lazy token bucket shaper with $B = 0$. Therefore, it enforces a simple rate control with no burstiness being allowed. It is described using the following equation:

$$\sigma(\Delta t) = \begin{cases} 0 & \text{if } \Delta t \leq 0, \\ \lceil R \cdot \Delta t \rceil & \text{else.} \end{cases} \tag{6.9}$$

The profile adherence test can be done using the simplified test described in Section 6.2.2, as the traffic profile is a simple periodic step function.

Full Refill. The *Full Refill* (FR) shaper is very similar to the token bucket shaper, yet in a simplified way. While in the token bucket algorithm, a single new token is added at a definable rate, the bucket is always completely refilled in the full refill algorithm. Its traffic profile function is described as follows:

$$\sigma(\Delta t) = \begin{cases} 0 & \text{if } \Delta t \leq 0, \\ B \cdot \lceil R \cdot \Delta t \rceil & \text{else.} \end{cases} \tag{6.10}$$

Every R^{-1} time units, the token bucket is refilled to its maximum of B tokens. The traffic profile function $\sigma(\Delta t)$ is a periodic step function profile as described in Section 6.2.2, therefore the profile adherence check is performed using the simplified check.

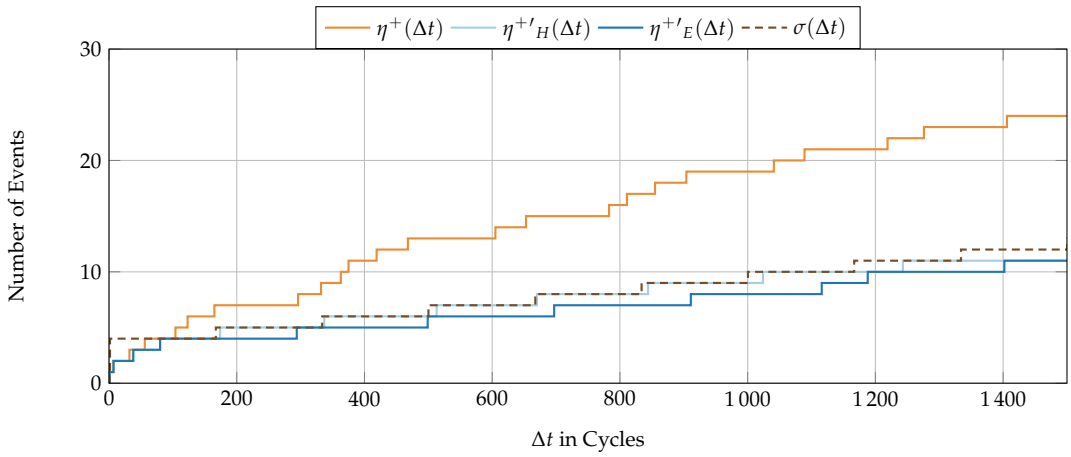


Figure 6.5. – Event arrival function of benchmark `qurt` shaped using a Lazy Token Bucket ($B = 3$, $R = 6/1\,000$) shaper.

6.5.2 Case Study

In the following, the implemented shapers and the two presented shaping approaches are illustrated for an exemplary benchmark. For this, the benchmark `qurt` from the MRTC benchmark suite [GBEL10] is chosen. This choice is arbitrary and does not focus on a specific feature of this benchmark. For the purpose of this case study to illustrate the shapers and shaped event arrival functions, a simple ARM7 single-core architecture (hence no differentiation between private or shared memories) is assumed. A more thorough evaluation on a broad set of benchmarks in a multi-core environment follows in the next section.

Figure 6.5 shows a detail of `qurt`'s unmodified upper event arrival function $\eta^+(\Delta t)$. The figure only shows the event arrival function up until a maximum interval length of 1 500 cycles to show a greater level of detail. The unmodified upper event arrival function converges with a total of 60 events at an interval length of approx. 7 200 cycles. Exemplary, each access to the `.data` section is assumed to generate an event. The instruction memory is assumed to have a latency of 6 cycles, hence the smallest delay μ by using a single delaying instruction is set to 6 cycles. All following shaper parameters are set to reduce the value of the upper event arrival function $\eta^+(\Delta t)$ at Δt = 1 000 cycles from originally 19 by more than 50% down to at most 9.

Figure 6.5 shows the traffic profile function $\sigma(\Delta t)$ for the lazy token bucket shaper, as well as the modified event arrival functions after shaping. As can be seen, the unmodified event arrival function $\eta^+(\Delta t)$ surpasses the traffic profile function $\sigma(\Delta t)$ at Δt = 104 cycles. $\eta^{+\prime}{}_E(\Delta t)$ is the resulting upper event arrival function after shaping the program with the evolutionary algorithm, whereas $\eta^{+\prime}{}_H(\Delta t)$ represents the curve after shaping using the greedy heuristic. The greedy heuristic's threshold ϵ is set to 0.2, as this enables a certain speedup of the algorithm (since the upper and lower bound of delay during the binary search are allowed to differ up to 20%), yet still forces the algorithm to cut down the delay added to a basic block using the binary search. The evolutionary algorithm was set to 20 generations and a population size of 10 individuals. Both shaping approaches result in event arrival functions which adhere to the traffic profile function. The resulting event arrival function of the greedy

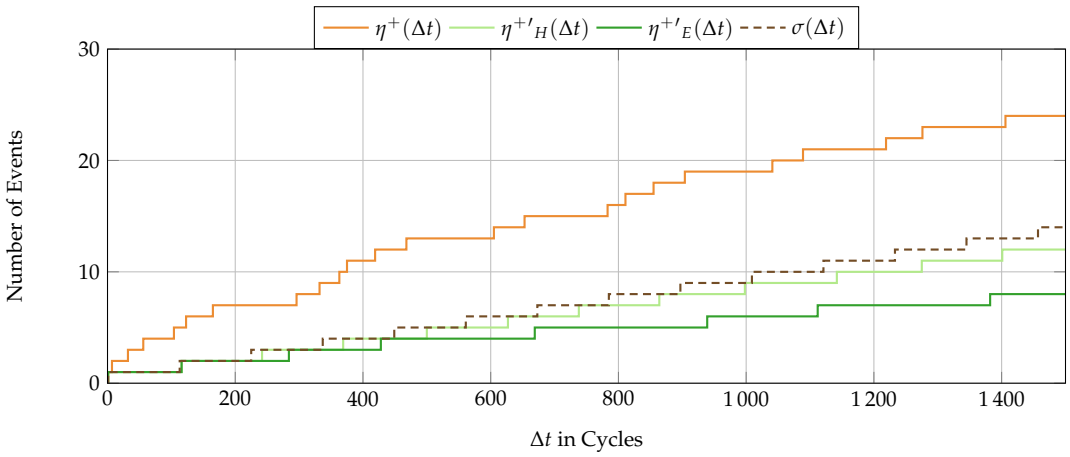


Figure 6.6. – Event arrival function of benchmark `qurt` shaped using a Lazy Rate ($R = 1/112$) shaper.

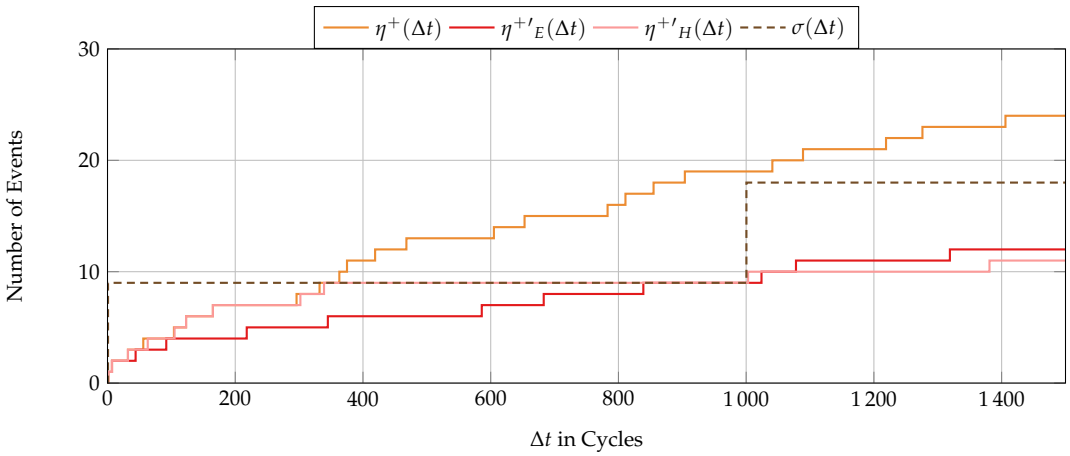


Figure 6.7. – Event arrival function of benchmark `qurt` shaped using a Full Refill ($B = 9$, $R = 1/1\,000$) shaper.

heuristic $\eta^{+\prime}{}_H(\Delta t)$ follows the traffic profile function very closely and is therefore close to the behavior of a greedy shaper in the case of lazy token bucket shaping. While $\eta^{+\prime}{}_E(\Delta t)$ is always less than or equal to the traffic profile function as well, it follows the profile function less close, potentially inserting more delay than necessary.

The traffic profile function $\sigma(\Delta t)$ used for the lazy rate shaper as well as both modified event arrival functions can be seen in Figure 6.6. As shown in the figure, the traffic profile function only allows one event every 112 cycles. Therefore, the unmodified event arrival function surpasses the traffic profile function already at $\Delta t = 7$ cycles with the minimum time between 2 accesses being 7 cycles. In general, the behavior of both shaping strategies is similar the previous shaper. While both modified event arrival functions are always below or equal to the traffic profile function, the function shaped using the greedy heuristic follows $\sigma(\Delta t)$ more closely.

Figure 6.7 shows the traffic profile function $\sigma(\Delta t)$ for the full refill shaper used in this example, as well as the unmodified upper event arrival function $\eta^+(\Delta t)$ and the shaped event arrival functions. Due to the higher initial step compared to the

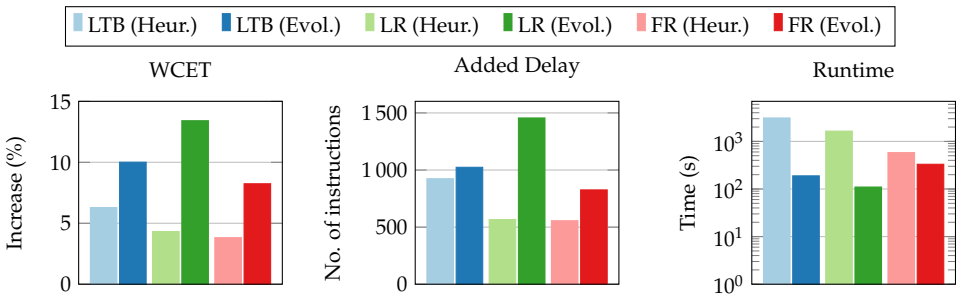


Figure 6.8. – Exemplary comparison of shapers for the `qurt` benchmark.

previous lazy token bucket shaper, the original event arrival function surpasses the traffic profile function first time at $\Delta t = 363$ cycles. Again, both modified event arrival functions adhere to the traffic profile function $\sigma(\Delta t)$. Similar as before, the event arrival function derived from the program shaped using the greedy heuristic ${\eta^+}'_H(\Delta t)$ follows the traffic profile function closely up to $\Delta t = 1\,000$ cycles. As can be seen, the greedy heuristic changed the upper event arrival function for the non-violating part ($\Delta t < 363$ cycles) only minimally. The event arrival function shaped by the evolutionary algorithm ${\eta^+}'_E(\Delta t)$ shows a greater deviation from the unmodified event arrival function $\eta^+(\Delta t)$ already for smaller values of Δt when compared to the heuristic. The evolutionary algorithm tends to distribute the delay among many basic blocks due to mutation and recombination, whereas the heuristic tends to choose a few basic blocks where most of the delay is added. This causes the plateau of the modified event arrival function ${\eta^+}'_H(\Delta t)$ of the heuristic between approx. $\Delta t = 380$ and $\Delta t = 1\,000$ cycles. The vast majority of delay is here simply placed in a few basic blocks which separate a maximum of 9 and 10 sequential events. While this allows the modified event arrival function ${\eta^+}'_H(\Delta t)$ of the heuristic to be very close to the profile function up to an interval length of 1 000 cycles, it also has the side-effect of the event arrival function ${\eta^+}'_E(\Delta t)$ of the EA slightly surpassing it for larger values of Δt. Here, the wider distribution of delay among the basic blocks by the evolutionary algorithm actually results in some events being less delayed when compared to the heuristic.

Figure 6.8 shows a comparison of three characteristics of the different shapers applied with the two different shaping strategies for the exemplary `qurt` benchmark. The profile shaping functions $\sigma(\Delta t)$ for the shapers correspond to the same as shown in the previous figures. Note that here a single-core system with only `qurt` running is regarded, hence no improvements in, e.g., WCRT for concurrent cores can be seen. This example only shows how efficiently the shapers and shaping strategies can modify a program to meet a considerably strong specification (reduce the max. number of events in a time interval of 1 000 cycles by more than 50%).

The largest increase in WCET after shaping can be seen for the lazy rate shaper implemented by the evolutionary algorithm with an increase of $\approx 13\%$. The lowest increase in WCET is achieved by the full refill shaper implemented with the greedy heuristic, increasing the WCET by only $\approx 4\%$. In this particular example, the greedy heuristic always yields a lower WCET increase for the same shaper when compared to the evolutionary algorithm.

The comparison of the added delay among the shapers and strategies shows a similar picture as for the increase in WCET. The lazy rate shaper results in the largest amount of delay added when implemented using the evolutionary algorithm. The least amount of delay is added when using the full refill shaper and the greedy heuristic.

Figure 6.8 also shows a comparison of the required runtime to shape the event arrival function when using the corresponding shaper and implementation strategy. As the number of individuals and generations are set equally for all executions of the evolutionary algorithms, the difference of runtimes among the EAs mostly show the different times required to perform a profile adherence check in this example. The smallest runtime required overall is achieved by the lazy rate shaper implemented by the evolutionary algorithm. This shaper uses the simple adherence check where only the maximum number of events $\eta^+(P)$ during a time interval with a length of the shaper's period P needs to be derived. For this exemplary program and the evolutionary algorithm, the lazy token bucket shaper ranks second and the full refill shaper the last in terms of runtime. This is interesting since the full refill shaper uses the simple adherence check, whereas the lazy token bucket shaper uses the more complex check. Although the ILP formulation is slightly more complex for the lazy token bucket due to the additional description of the profile function, it takes on average less time to solve than compared to a single adherence check of the full refill shaper. This is due to the comparably larger interval Δt to be checked for the full refill shaper. When compared to the lazy rate shaper (both use the simplified check), the adherence check for the lazy rate shaper needs to calculate the value of $\eta^+(112 \text{ cycles})$, whereas for the full refill shaper the value of $\eta^+(1\,000 \text{ cycles})$ needs to be derived (as those are the shapers' periods). This easily increases the potential solution space, as the number of potential sub-paths with a maximum execution time of at most 1 000 cycles is larger than if a sub-path is only allowed to take at most 112 cycles, potentially increasing the solving time for the full refill shaper. In fact for this particular example, this larger time interval increases the average time for performing a single adherence check for the full refill shaper above the average time for the adherence check of the lazy token bucket shaper. This shows that the more complex ILP description does not necessarily lead to longer solving times. Yet again, this is very depending on the individual program and shaper parameters.

When comparing the runtimes of the greedy heuristic with the evolutionary algorithm, the former always takes longer compared to the evolutionary algorithm when implementing the same shaper in this example. Particularly the lazy token bucket shaper takes 1 644% longer to implement with the greedy heuristic compared to the evolutionary algorithm.

Figure 6.9 shows the convergence of the evolutionary algorithm in terms of WCET in this example. For all three evaluated shapers, at least one individual already does not violate the traffic profile function in the very first generation. This is most likely due to the one heavily modified individual which is placed in every initial population. This individual leads to a solution adhering to the traffic profile function, yet at the cost of a large WCET increase. Therefore, the WCET value of the best individual is considerably large at the first generation for all three shapers. In the third generation, a significantly better solution is found for all shapers. Subsequently, the solution

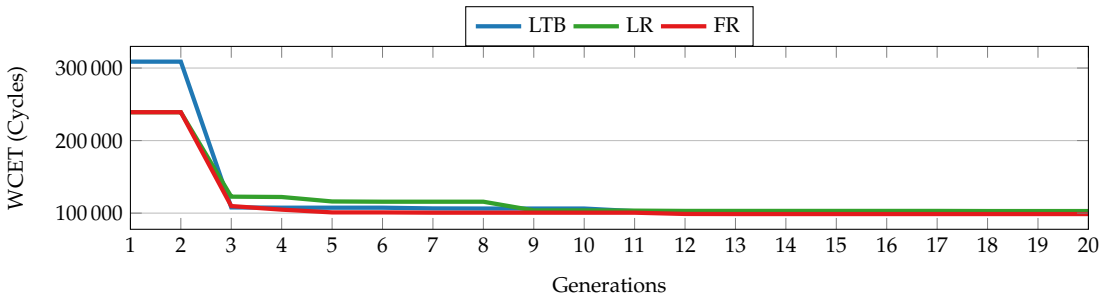


Figure 6.9. – Lowest WCET with no traffic profile function violation per generation when using the evolutionary algorithm.

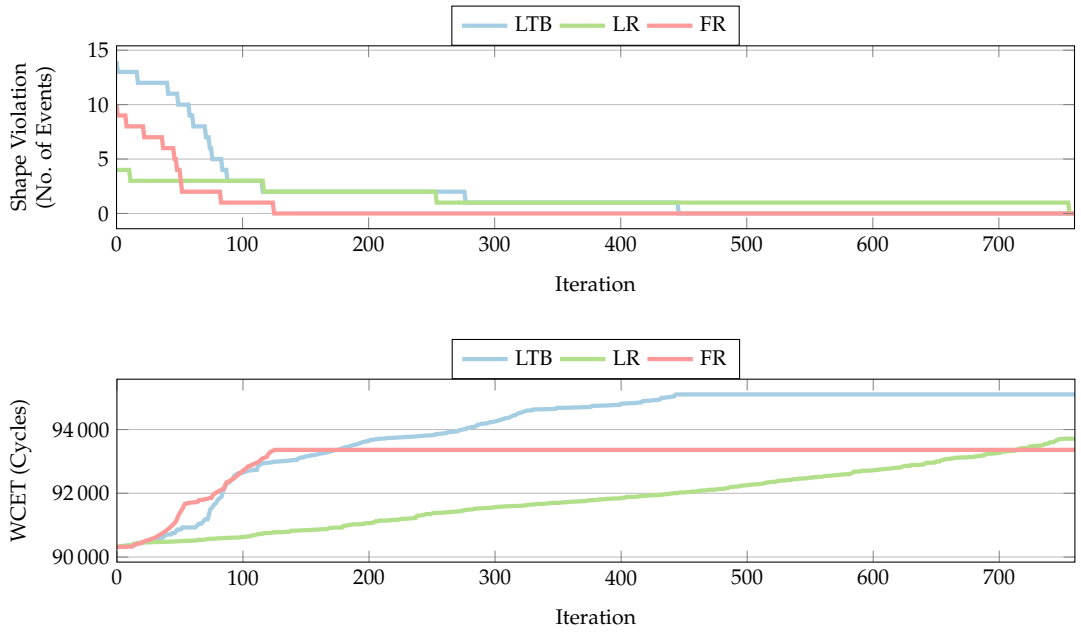


Figure 6.10. – Traffic profile function violation and the corresponding WCET per iteration of the greedy heuristic.

only marginally improves and converges close to its final value after 9 generations, whereas the full refill shaper has the fastest convergence in terms of generations.

Similarly, Figure 6.10 shows the traffic profile function violation (upper figure) and the corresponding WCET (lower figure) per iteration and shaper for the greedy heuristic. The greedy heuristic terminates as soon as the traffic profile violation reaches 0. As can be seen, the greedy heuristic can implement the full refill shaper with the least number of iterations (125), whereas the lazy rate shaper takes the largest number of iterations (756). The significantly greater number of required iterations is likely due to the larger number of violating paths to modify. As the lazy rate shaper only allows one event every 112 cycles in this example, *every* 2 consecutive events with less than 112 cycles in between them is a sub-path violating the traffic profile function. The differing number of initial violating events is due to the different profile adherence checks and shapers. As the adherence check for the full refill and lazy rate shapers are implemented by simply checking the value of $\eta^+(\Delta t)$ at their period, the initial profile function violation corresponds to the difference of $\eta^+(\Delta t) - \sigma(\Delta t)$ at this

Δt value. For the lazy token bucket shaper, the adherence check is implemented by describing $\sigma(\Delta t)$ inside the ILP model and maximizing the difference $\eta^+(\Delta t) - \sigma(\Delta t)$. Therefore, the initial traffic profile violation corresponds to the greatest difference between the unmodified event arrival function and $\sigma(\Delta t)$ of the lazy token bucket shaper.

The lower figure of Figure 6.10 shows the evolution of the WCET over the heuristic's iterations for each shaper. The steepest slope can be seen for the full refill shaper, yet as it terminates the earliest, it actually has the lowest overall increase in WCET. In contrast, the lazy rape shaper has the flattest slope in terms of WCET increase per iterations and takes the largest number of iterations.

Overall, it should be noted that these figures and results only illustrate one specific example. All features of the three presented shapers and both implementation strategies are depending on the actual traffic profile function to be implemented and the actual program to be shaped.

6.5.3 Use Case

A possible use case for applying the presented methods of WCET-aware code-inherent traffic shaping is to reduce the WCRT of certain tasks inside a multi-core system. Especially if one task is meeting its deadline with a certain slack, while another task on another core is not meeting is deadline yet, the first task could be shaped to reduce the bus contention and reduce the other task's WCRT. This potential use case is utilized for a more thorough evaluation of the two different shaping methods and the three implemented exemplary shapers introduced previously.

The assumed multi-core architecture resembles the exemplary base architecture shown in Figure 2.6 (cf. Page 13) with a single task allocated to each core. A single core is assumed to be an ARM7TDMI processor. As the presented methods expect the code to be shaped not being placed inside a shared memory (as the delaying code would then in turn create further bus contention), it is assumed that the private instruction SPM of each core is large enough to hold the allocated task's code. The data objects of each task remain in the shared memory and therefore require a bus access. Round robin-based and fixed priority-based bus arbitration are evaluated in two different scenarios. For both policies, a non-preemptive bus arbitration is assumed, meaning once a core was granted access to the bus, this access cannot be interrupted by another core. Multi-core systems with 2, 4 and 8 cores are evaluated. For each system, one task with an expected high influence on the other cores is chosen to be shaped. In case of a multi-core system with a fixed priority-based bus arbitration, the task allocated to the core with the highest bus priority is chosen to be shaped. For systems with a round robin-based bus arbitration, the task with the highest number of bus accesses (normalized to the task's period) is shaped. For each system, shaping method and shaper profile, relative event reductions of 5%, 10%, 15% and 20% at 1 000 cycles are evaluated. In case of a 5% reduction, this means that the shaper has to reduce the maximum number of bus accesses in a time interval of 1 000 cycles of the to be shaped task by at least 5%. The fixed time interval of 1 000 cycles is chosen arbitrarily and is simply used as a time window independent from task characteristics.

The shared memory is assumed to have an access latency of 6 cycles (similar to the default setting of existing embedded systems [NXP09a, Inf07]), whereas an SPM access is assumed with a single cycle latency. Each task has a periodic activation

where the task's period is set to two times its WCET (leading to a utilization of 50% per core in isolation). Benchmarks of the following benchmark suites were evaluated: MRTC [GBEL10], MediaBench [LPM97], StreamIt [Str18], and UTDSP [LCS92]. Additionally, a set of miscellaneous benchmarks mostly consisting of de- and encoders were evaluated as well. Benchmarks from this set were randomly ordered and grouped by using a sliding window over the list. This results in a total number of 86 different systems to be evaluated for each multi-core setting. See Appendix E.2 for a detailed overview of all evaluated benchmark configurations. In total, more than 12 300 different settings are evaluated (dual-, quad- and octa-core architectures, 86 benchmark configurations per architecture, fixed priority- and round robin-based bus arbitration, 4 different event reduction values for 3 different shapers, each implemented once using the greedy heuristic and once via the evolutionary algorithm).

All experiments were performed on an Intel Xeon Server (48 cores at 3.2 GHz with 1.48 TB RAM) and ILPs were solved using Gurobi 8.1.0, whereas each ILP solving process was limited to 1 thread. All benchmarks were compiled with several ACET-oriented optimizations activated (`-O2` optimization flag of the WCC compiler, cf. Chapter 3). A general timeout of 1 h per core is used for the optimization.

Timing Analysis

For this evaluation, a so-called *semi-integrated* or *semi-joint* multi-core timing analysis is used. As the name suggests, this analysis approach is placed in between a full joint multi-core WCRT analysis (where the microarchitectural analysis is performed for all cores in parallel) and a fully compositional timing analysis (where timing effects due to shared resources are derived completely separated from the task's WCET and are then simply added). The methods used for this timing analysis are based on the principles presented by Jacobs et al. [JHH15]. Ideally, this timing analysis method tries to combine the advantages of a joint multi-core timing analysis and a fully compositional one, namely a high precision, possible support for architectures with timing anomalies and good scalability. This timing analysis is chosen over the fully compositional timing analysis used in the previous evaluation for the extraction of event arrival functions[1] (cf. Section 5.6) as it is more precise and does not require to derive complete event arrival functions, but only single values.

The basic idea is to derive upper bounds on the maximum number of cycles a task may be blocked from accessing a shared resource in total on a system-level, and to integrate this upper bound into the low-level path analysis. The maximum number of cycles a task may be blocked from accessing a shared resource can be derived using the event arrival functions of the tasks allocated onto the concurrent cores as shown in the equations in Section 5.6.1 (cf. Page 114). The IPET [LM95] path analysis approach is adapted to support variable access latencies for all shared memory accesses, whereas the total number of blocked cycles can be upper-bounded. By performing an initial path analysis where each bus access is assumed with the worst-case timing, a safe, yet potentially pessimistic upper bound on the WCRT is found. Subsequently, an upper bound on the total number of stall cycles for the task under analysis is derived by calculating the maximum number of bus accesses of the

[1] The previous evaluation used the fully compositional timing analysis, as event arrival functions are most commonly used in these timing analyses.

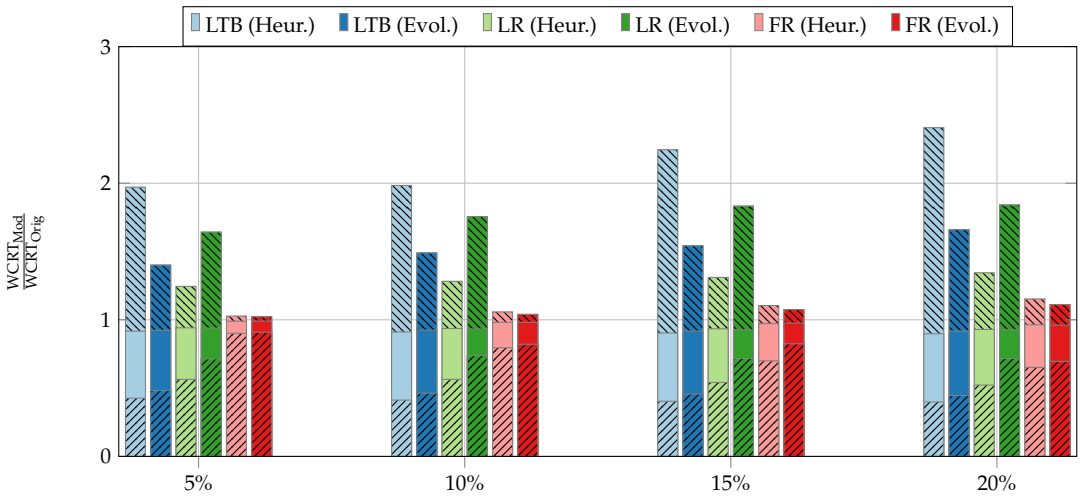


Figure 6.11. – Average normalized WCRT after traffic shaping of the shaped task (▧), the unshaped tasks (□) and the minimum normalized WCRT (▨) for dual-core systems with a fixed-priority bus arbitration policy, depending on the minimum amount of events to be reduced, traffic shaping profile and shaping implementation method.

concurrent cores during a time interval of length equal to the previously determined WCRT. This upper bound on the total number of stall cycles is then in turn inserted in the path analysis ILP to derive a more precise WCRT. This is done iteratively until a stable WCRT is found. As this approach does not rely on any external analysis tools, the event arrival functions do not need to be approximated beforehand, but each value of the event arrival function can be calculated and cached as it is required. This timing analysis approach is used instead of a purely compositional one, as it can significantly improve the precision of the worst-case timing to be derived while still offering scalability.

Dual-Core Evaluation

Figure 6.11 shows the evaluation results for the dual-core systems with a fixed-priority-based bus arbitration. The x-axis lists the amount of maximum bus accesses to be reduced from the task allocated on the highest priority core in a time interval with length of 1 000 cycles. The y-axis is the normalized WCRT of a task, whereas $\mathrm{WCRT}_{\mathrm{Mod}}$ is the WCRT after the traffic shaping was applied and $\mathrm{WCRT}_{\mathrm{Orig}}$ is the initial WCRT. Figure 6.11 shows 3 bar plots behind each other: The lowest bars (marked with ▨) represent the overall lowest normalized WCRT observed for each specific configuration, corresponding to the largest relative WCRT decrease due to the traffic shaping. The middle bars (marked with no hatching) represent the average (geometric mean) normalized WCRT of all unshaped tasks inside a system. For the dual-core systems, this corresponds to the average normalized WCRT of all tasks allocated to the lower priority core of the system. The last bar (marked with ▧) represents the average (geometric mean) normalized WCRT of all shaped tasks. Note, that the bars are *not* stacked on each other, but simply plotted behind each other. Therefore, the value of each bar can be read using the absolute y-axis scale.

As expected, the WCRT of the task whose bus access profile was shaped is increasing on average with the amount of bus accesses to be reduced. Although both implementation approaches are WCET-aware, it is not always possible to exploit the specific control-flow graph of a task to apply traffic shaping without any impact on the task's WCRT. This negative effect can be seen very dominantly for the lazy token bucket (LTB) shaper when implemented with the greedy heuristic. Here, the average WCRT is increased by a factor of 1.97 for a reduction of bus accesses by 5%, up to an average WCRT increase by a factor of 2.41 for a bus access reduction of 20%. Overall, large differences can be seen for the WCRT increase of the shaped task between the different shapers and even between individual implementation methods. The full refill (FR) shaper here shows the overall best performance, as it continuously results in the lowest average WCRT increase. The greedy heuristic and the evolutionary-based approach deliver nearly identical results for the full refill shaper, yet the evolutionary-based approach leads to a slightly smaller WCRT in most cases. Even for a comparably large reduction of bus accesses of 20% in the given time interval, the full refill shaper increases WCRT only by a factor of 1.11 on average. It is also noticeable that some shapers are significantly more efficient implemented with either the greedy heuristic or the evolutionary-based approach. When implementing the lazy token bucket shaper with the greedy heuristic instead of the evolutionary-based approach, the WCRT increases on average by 62%. On the other side, the lazy rate (LR) shaper is on average better implemented using the greedy heuristic instead of the evolutionary-based approach. Here, the WCRT increases on average by 47% when using the evolutionary-based approach. In case of the lazy rate shaper, potentially many sub-paths are initially violating the access profile and have to be slightly adjusted, as the step function always only increases by a single event. Therefore, a minimum distance between two single accesses is enforced. This large amount of sub-paths to be fixed, while each sub-path potentially only requires a small amount of delay, easily slows down the convergence of the evolutionary approach. In case of the lazy token bucket (LTB) shaper, the combination of the greedy heuristic and the specific profile adherence check might not be ideal. For the lazy token bucket shaper, the profile adherence check is done by describing the access profile within the event arrival function ILP model and maximizing the difference between those two curves (cf. Section 6.2.2). The resulting violating sub-path is then adjusted by the greedy heuristic. As this violating sub-path may change in the next iteration when another sub-path becomes the most violating one (similar to a WCEP switch), the greedy heuristic potentially jumps between adjusting many different sub-paths. This less-focused behavior may leave superfluous delaying instructions behind, resulting in a higher WCRT increase.

These results on the average increase in WCRT of the shaped task are partially contrary to the results seen in the case study of the single benchmark `qurt` (cf. Figure 6.8, Page 146). There, using the greedy heuristic for implementing the lazy token bucket shaper was more efficient in terms of WCET increase compared to the evolutionary algorithm. Yet, for the average over all evaluated benchmarks, the relation is inverse. Similarly, the full refill shaper implemented using the evolutionary algorithm resulted in a greater WCET increase compared to the greedy heuristic for the single benchmark case, while also this relation is reversed when regarding the average results in Figure 6.11. These contrary outcomes indicate that although the average results over

many benchmarks can display an overall impression of the performance of the shaper and implementation method combinations, the exact impact can be highly depending on the individual program. This can be caused by the total number and distribution of bus accesses, as well as the control-flow structure of the program. These parameters influence the required time for an adherence check and whether the greedy heuristic or evolutionary algorithm suits better.

Overall, the average normalized WCRT of the unshaped tasks (marked with no hatching) decreases as expected with a larger amount of bus accesses reduced. E.g., the average normalized WCRT decreases to 0.991 when the task allocated on the highest priority core is shaped using the full refill shaper implemented with the EA-based approach and a bus access reduction of 5%. When increasing the bus access reduction to 20%, the average normalized WCRT decreases to 0.961. Furthermore, mostly complementary results for the average normalized WCRTs of the unshaped tasks can be seen when compared to the results of the shaped tasks. While the lazy token bucket shaper implemented with the greedy heuristic consistently leads to the largest WCRT *increase* for the shaped task, it also leads to the overall largest average WCRT *decreases* for the remaining tasks. This is most likely due an overly aggressive shaping where more bus accesses than actually required are delayed. While this leads to a stronger increase in WCRT for the shaped task in most cases, it also leads to a lower WCRT for the remaining tasks, as the number of competing bus accesses per time interval decreased. The lazy rate shaper falls in between the full refill shaper and the regular lazy token bucket shaper in terms of average WCRT reduction for the unshaped tasks.

When regarding the overall minimum normalized WCRTs (marked with ▨), large reductions can be seen. Using the lazy token bucket shaper implemented with the greedy heuristic results in WCRT reductions of up to 60.1% (corresponding to a normalized WCRT of 0.399). But also the shapers or implementation methods with a lower impact on the shaped task's WCRT can reach a significant WCRT reduction. E.g., the largest WCRT reduction observed when using the full refill shaper implemented with the evolutionary algorithm range from a factor of 0.91 for a bus access reduction of 5%, down to 0.696 for a bus access reduction of 20%.

Figure 6.12 shows the results for the dual-core systems evaluated with a round robin-based bus arbitration policy. The structure of the plot is identical to the previous evaluation for the systems with a fixed-priority-based bus arbitration. The overall distribution of the different shaper and implementation method results for the average normalized WCRT of the shaped task (marked with ▧) are very similar to the previous evaluation. The lazy token bucket shaper when implemented with the greedy heuristic consistently performs the worst in regard of the average WCRT increase for the shaped task, while the full refill shaper implemented with the evolutionary-based approach performs best. Similarly, the average normalized WCRT of the shaped task increases with a larger amount of bus accesses per time interval to be reduced for all shapers and implementation methods. Yet, the general WCRT increase for the shaped task is slightly higher for the dual-core systems with a round robin-based bus arbitration than compared to fixed-priority-based bus arbitration. A possible reason for this is the lower possible blocking for the shaped task in the fixed-priority-based multi-core bus system than in the round robin-based one. The delaying instructions may not only increase the WCRT by the additional delay on the

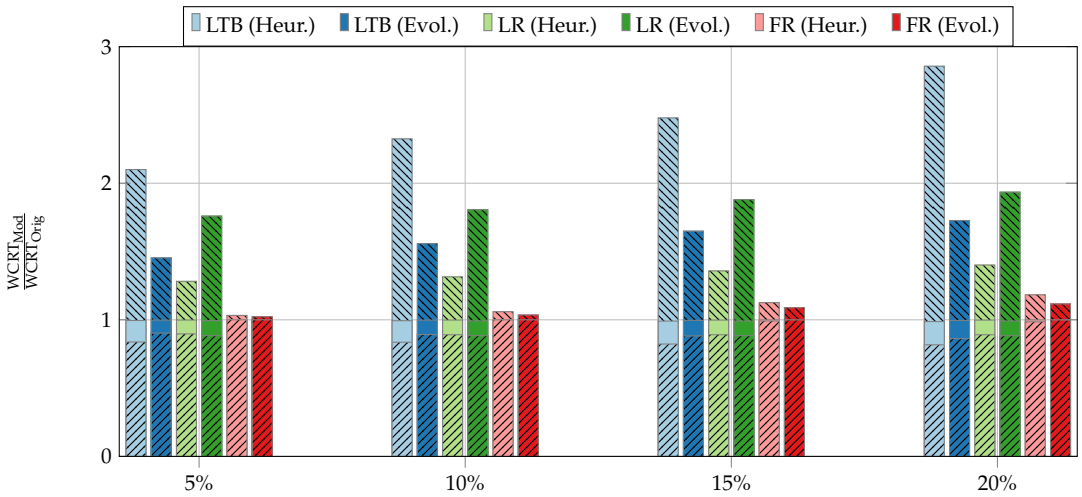


Figure 6.12. – Average normalized WCRT after traffic shaping of the shaped task (▨), the unshaped tasks (□) and the minimum normalized WCRT (▨) for dual-core systems with a round robin bus arbitration policy, depending on the minimum amount of events to be reduced, traffic shaping profile and shaping implementation method.

WCEP, but by increasing the WCET of a task, the number of potentially competing bus accesses is also increasing. Since all cores have the same bus priority inside a round robin-based bus system, while the task which is shaped in the fixed-priority-based bus system is allocated to the highest priority core, the penalties for an increased number of competing bus accesses differ.

The average normalized WCRTs for the unshaped tasks (marked with no hatching) are only affected very marginally by the shaping of the other task in the system. While the average WCRT reduction for these tasks is also increasing for a larger amount of bus accesses reduced per time interval, this average reduction is significantly smaller than compared to the results from the fixed-priority-based bus systems. For systems shaped with the lazy token bucket shaper, implemented with the greedy heuristic, the average normalized WCRT ranges from a from 0.995 for a bus access reduction of 5%, down to 0.987 for a bus access reduction of 20%. In general, the degree of WCRT reduction by shaping a single task is expected to be greatly lower for architectures with a round robin-based bus arbitration than compared to architectures with a fixed-priority-based arbitration. This is due to the tight upper bound per single access for a round robin-based bus. In case of a dual-core system, a single access can only be delayed by at most the length of another access, as bus priorities are rotated after each access. Additionally, the reduction of events may not have any effects on another task accessing the bus.

Example 6.1. *Given a system with two cores with a round robin-based bus arbitration, it is assumed that a task τ_0 is allocated to one core, whereas a second task τ_1 is allocated to the other core and has a maximum number of bus accesses of $A^+_{E,1}$ during one execution. Task τ_1 has an initial WCRT of R^+_1. The bus access profile of task τ_0 is being shaped from originally $\eta^+_0(R^+_1) = 500$ to $\eta^{+\prime}_0(R^+_1) = 400$. In case the maximum number of bus accesses during one complete execution of task τ_1 is lower than or equal to 400 ($A^+_{E,1} \leq 400$), the WCRT of task τ_1 is not lowered due to the shaping of task τ_0. As the maximum number of bus accesses of*

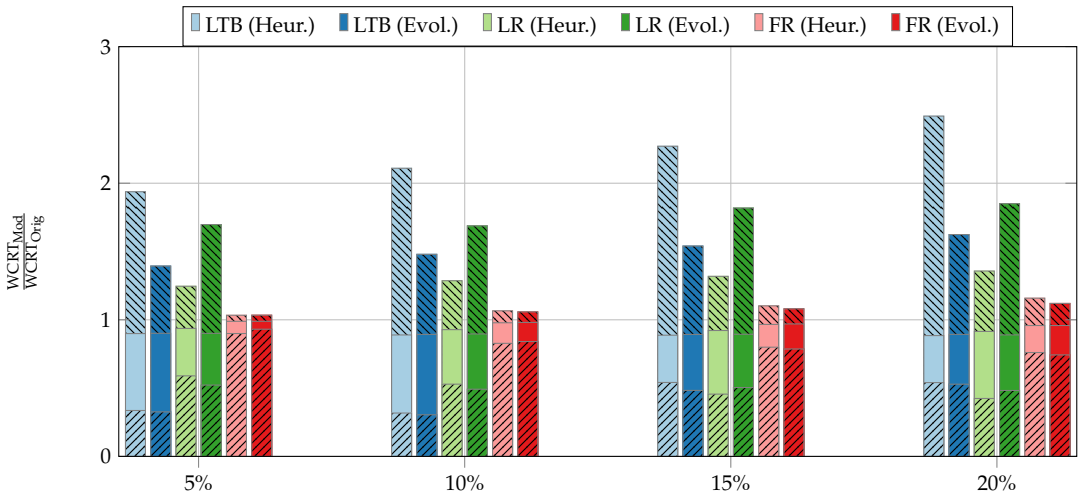


Figure 6.13. – Average normalized WCRT after traffic shaping of the shaped task (▧), the unshaped tasks (□) and the minimum normalized WCRT (▨) for quad-core systems with a fixed-priority bus arbitration policy, depending on the minimum amount of events to be reduced, traffic shaping profile and shaping implementation method.

task τ_1 being blocked is $\min(\eta_0^+(R_1^+), A_{E,1}^+)$*, the maximum number of bus accesses of task τ_0 in a time interval of length R_1^+ has to become lower than $A_{E,1}^+$, such that task τ_0's WCRT will actually become affected by the traffic shaping.*

Following this trend, the minimum normalized WCRTs observed for the dual-core systems with a round robin-based bus arbitration are higher than compared to the fixed-priority bus counterparts, yet still some significant decreases can be noticed. Using the lazy token bucket shaper implemented with the greedy heuristic, the minimum normalized WCRT observed is 0.817. For the less intrusive combination of the lazy rate shaper with the greedy heuristic, the lowest normalized WCRT is 0.891.

Quad-Core Evaluation

Figure 6.13 shows the results for the quad-core systems with a fixed-priority-based bus arbitration. As before, the structure of the plot is identical to the ones discussed previously. Overall, the distribution of the different timings for the shapers and implementation methods are very similar to the results of the dual-core systems with a fixed-priority-based bus (cf. Figure 6.11). Focusing on the average normalized WCRT of the shaped task (marked with ▧), the lazy token bucket shaper implemented with the greedy heuristic consistently results in the worst performance. Its outcome ranges from an average WCRT increase by a factor of 1.938 for a bus access reduction of 5% up to a WCRT increase by a factor of 2.501 for a bus access reduction of 20%. The full refill shaper, especially when implemented using the evolutionary-based approach also yields the best results here with an average WCRT increase by factor of 1.034 for the lowest amount of bus access reduction and by a factor of only 1.12 for the highest one. In general, the average normalized WCRT of the shaped tasks slightly increased when compared to the results of the dual-core systems with a fixed-priority-based bus arbitration with about 0.77%. This slight increase can be explained

due to the additional number of cores in the system. Although the task whose bus access profile is shaped is allocated to the core with the highest bus priority, the task may still experience slight bus blocking times in case a lower priority core initiated a bus access just before the higher priority core (cf. the min-term in Equation (5.143), Page 114). A larger number of cores in the system may therefore lead to a larger number of potentially competing accesses when the WCRT of a task is increased.

Also, the average normalized WCRTs of the unshaped tasks (marked with no hatching) in the quad-core systems with fixed-priority-based bus arbitration are very similar to the results of the dual-core evaluation. With an increasing amount of reduced maximum bus accesses, the average decrease in WCRT for the unshaped tasks grows. Overall, the average normalized WCRT among all shapers and implementation methods is slightly larger for the quad-core systems with 0.941 than compared to the dual-core systems with 0.925. This slight increase is due to the smaller ratio of tasks whose bus access profile was shaped and unshaped tasks. In case of a dual-core system with one task allocated to each core, the task on the lower priority core is only dependent on the bus access behavior of the high priority core. Whereas in a quad-core system, the task on the lowest priority core is dependent on the behavior of all 3 higher priority cores, from which only one core's behavior is shaped. Yet, although the average WCRT reduction per unshaped task slightly decreased with around 1.62%, the number of tasks profiting from the shaping increased from one to three tasks per system.

The overall minimum normalized WCRT (marked with ▨) observed for the quad-core systems with a fixed-prioty-based bus arbitration follow a contrary trend to the average values for the unshaped tasks. The lowest normalized WCRT observed for the dual-core systems was 0.399, whereas this value decreased further to 0.305 for the quad-core systems. A possible reason for the general decrease of the minimum normalized WCRT is the larger amount of positively affected tasks by the shaping of a single task. Therefore, the chance of finding a task which profits more by shaping the bus access behavior a specific task also increases. Yet, for some shaper, implementation method and bus access reduction combinations, the minimum relative WCRT observed increased (e.g., the full refill shaper, implemented with the evolutionary-based approach and a bus access reduction of 5%). This contrary observation can be explained with the overall larger runtime required for larger systems, leading to a higher chance of evaluations being canceled due to the timeout limit. Hence, a specific evaluation system combination leading to large WCRT increases for the dual-core system may be canceled due to a timeout in the quad-core evaluation, not resulting in any WCRT decrease. Nonetheless, still significant WCRT reductions are observed due to the shaping of the bus access profile of the highest priority core.

The results for the quad-core systems with a round robin-based bus arbitration are depicted in Figure 6.14. The relative distribution between the average normalized WCRTs of the shaped task (marked with ▧) is similar to the one seen for the dual-core systems (cf. Figure 6.12). In contrast to the trend seen for the systems with a fixed-priority-based bus, the average WCRT increase for the shaped task is lower for the quad-core systems with a round robin-based based than compared to the dual-core systems. This trend suggests that shaping a task has on average a smaller relative impact on the shaped task's WCRT inside a round robin-based bus system with an increasing number of cores. Whereas the average WCRT increase ranged from a factor

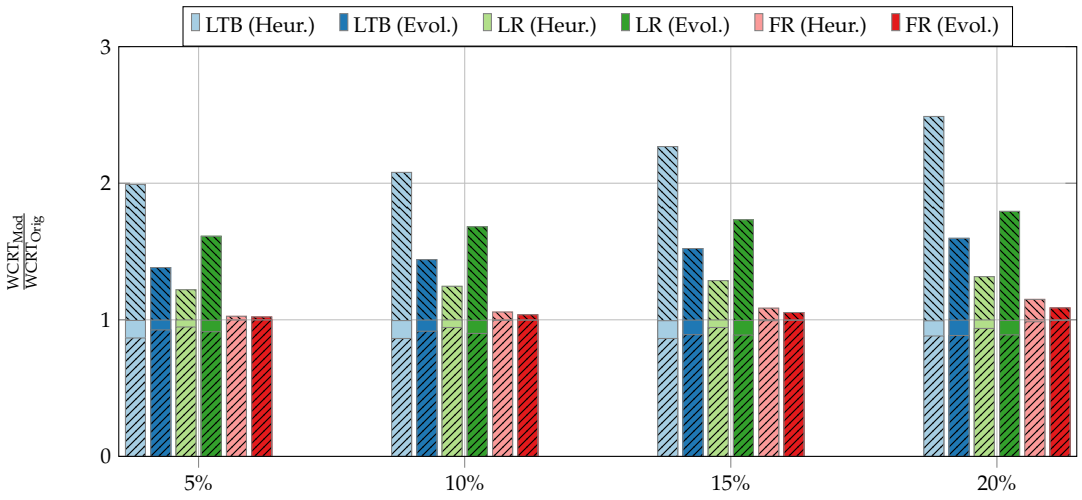


Figure 6.14. – Average normalized WCRT after traffic shaping of the shaped task (▨), the unshaped tasks (□) and the minimum normalized WCRT (▨) for quad-core systems with a round robin bus arbitration policy, depending on the minimum amount of events to be reduced, traffic shaping profile and shaping implementation method.

of 1.022 to 1.116 for the full refill shaper implemented with the evolutionary-based approach for the dual-core systems, this range dropped to 1.021 to 1.087 for the quad-core systems. Similarly, the largest average increase in WCRT due to the lazy token bucket shaper implemented with the greedy heuristic decreased from a factor of 2.857 at the dual-core systems to 2.486 at the quad-core systems. This overall reduction of average WCRT increase reduces the relative differences between the shapers and implementation methods.

As seen for the dual-core systems, the average normalized WCRT of the unshaped tasks (marked with no hatching) are very minor compared to the average WCRT reductions of the fixed-priority-based bus systems. The aggressive shaping enforced by the lazy token bucket shaper implemented with the greedy heuristic results in the lowest average normalized WCRT per task with 0.993 for a bus access reduction of 20%. Despite the considerably low average WCRT reduction, the results for the overall minimum normalized WCRT observed (marked with ▨) still show that in some cases, significant WCRT reductions can be achieved. Overall, the lowest normalized WCRT observed is 0.863, which relates to a WCRT decrease of 13.7%. Compared to the results of the dual-core systems with a round robin-based bus arbitration, the maximum WCRT reduction observed decreased for the quad-core systems. With a larger number of cores, the influence of a single core's bus access behavior being shaped reduces as well. Since the worst-case bus access latency is tightly bounded for a round robin-based bus arbitration (as the bus priorities rotate after each access), the best relative reduction of bus blocking times of a single task by shaping the behavior of a another task is bounded by about $\frac{1}{N_C}$.

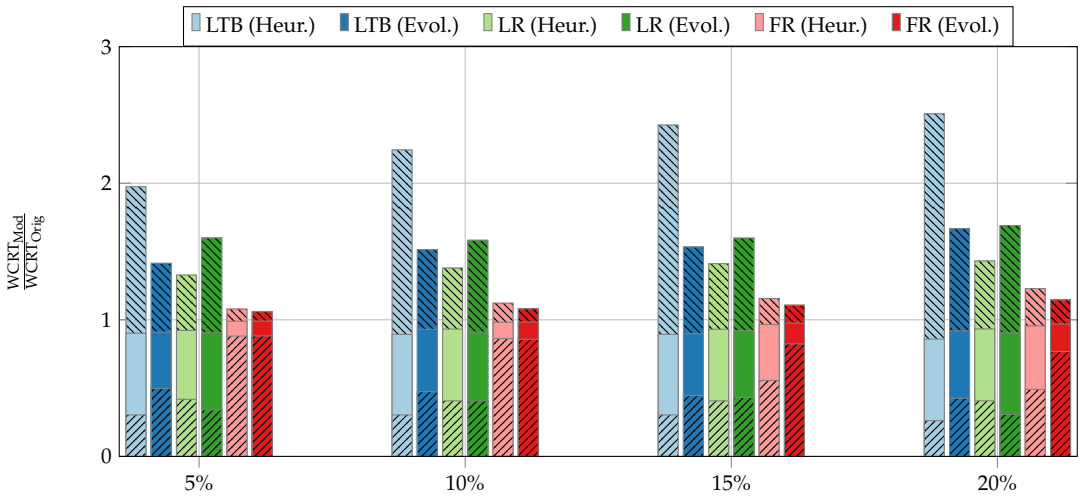


Figure 6.15. – Average normalized WCRT after traffic shaping of the shaped task (▨), the unshaped tasks (□) and the minimum normalized WCRT (▨) for octa-core systems with a fixed-priority bus arbitration policy, depending on the minimum amount of events to be reduced, traffic shaping profile and shaping implementation method.

Octa-Core Evaluation

Figure 6.15 shows the results for the octa-core systems evaluated with a fixed-priority-based bus arbitration policy. The trends observed from the change from a dual-core system to a quad-core one continue with the further addition of four more cores to the system. The overall relative distribution of the average normalized WCRT of the shaped task (marked with ▨) is very similar to the one of the quad-core or dual-core systems. The lazy token bucket shaper implemented with the greedy heuristic continues to show the worst performance for this characteristic, whereas the full refill shaper implemented with the evolutionary-based approach again generates the best results. As seen already for the quad-core systems, the relative WCRT increase for the shaped task grows on average when moving from a quad-core system to an octa-core system. While the average normalized WCRT of the shaped task was in the range of 1.03 to 2.49 for the quad-core systems, this range rises to 1.06 to 2.51 for the evaluated octa-core systems.

Similarly, the average WCRT decrease of the remaining tasks in the octa-core systems is slightly lower when compared to the quad-core systems. For the octa-core systems, an unshaped task's WCRT is reduced by 6.8% on average (geometric mean, among all shapers and implementation methods), while for the quad-core systems, this factor was at 7.5%. This is most likely due to the smaller influence of a single core with a reduced bus access behavior in a system with a rising number of cores. Yet, this decrease in the average WCRT reduction is not proportional with the additional number of tasks actually profiting of shaping the bus access behavior of the task allocated to the highest priority core, as *each* of the 7 task's WCRTs may decrease.

Regarding the minimum normalized WCRTs (marked with ▨) for the octa-core systems, the same trend as for the dual-core to quad-core systems can be noticed. Overall, the largest relative WCRT reductions observed further increased for the octa-core systems. While the lowest normalized WCRT observed for the quad-core

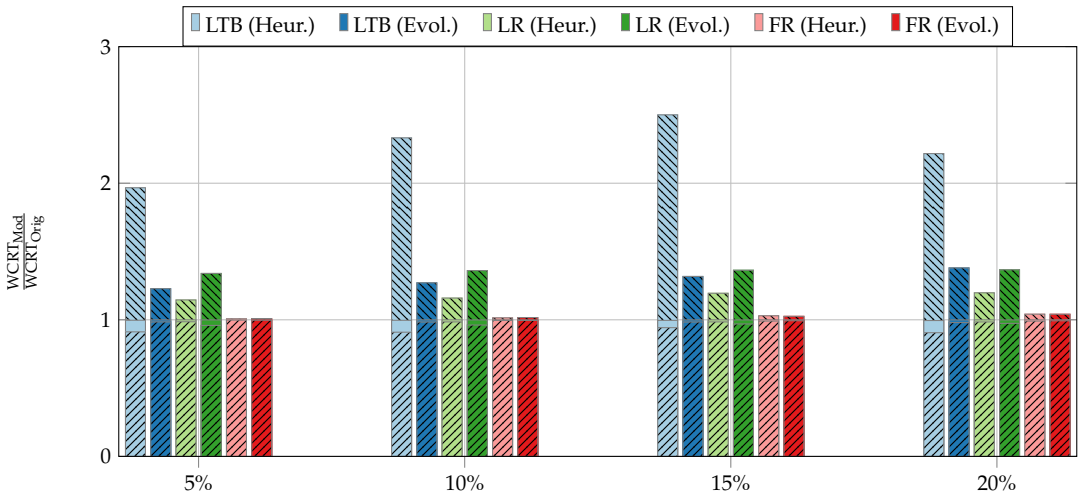


Figure 6.16. – Average normalized WCRT after traffic shaping of the shaped task (▧), the unshaped tasks (□) and the minimum normalized WCRT (▨) for octa-core systems with a round robin bus arbitration policy, depending on the minimum amount of events to be reduced, traffic shaping profile and shaping implementation method.

systems was 0.305, this further decreased down to 0.262 for the octa-core systems. On average, the largest WCRT reduction per shaper, implementation method and bus reduction increased by 7.68% when comparing the octa-core system results to the quad-core ones.

The evaluation results for the octa-core systems with a round robin-based bus arbitration are shown in Figure 6.16. Overall, the results follow the same trends as noticed for the quad-core systems. The average normalized WCRT of the shaped task (marked with ▧) further decreased compared to the evaluated quad-core systems. Among all shapers, implementation methods and bus access reductions, the WCRT increase of the shaped task was on average 11.16% lower for the octa-core systems when compared to the results of the quad-core systems. A notable anomaly can be seen for the lazy token bucket shaper implemented with the greedy heuristic for a bus access reduction of 20%. Here, the bar for the average normalized WCRT is *lower* compared to the bus access reduction value of 15%. This seemingly anomaly is due to the large runtime required for the lazy token bucket shaper implemented with the greedy heuristic in combination with the longer time required for the final timing analysis in an octa-core system, leading to more evaluations being canceled due to a timeout. In this case, this causes the bar for the lazy token bucket shaper implemented with the greedy heuristic at 20% to be lower than compared to the 15% reduction.

The average WCRT reductions of the unshaped tasks (marked with no hatching) are still very small and even decreased further when compared to the quad-core results. This trend can also be seen for the minimum normalized WCRTs (marked with ▨). Whereas the lowest normalized WCRT was 0.863 for the quad-core systems, it is 0.906 for the octa-core systems.

The average runtimes for performing the WCET-aware traffic shaping of a single task are depicted in Figure 6.17. These times only include the actual runtime for the shaping optimization, timings required for the parsing, code selection or final timing

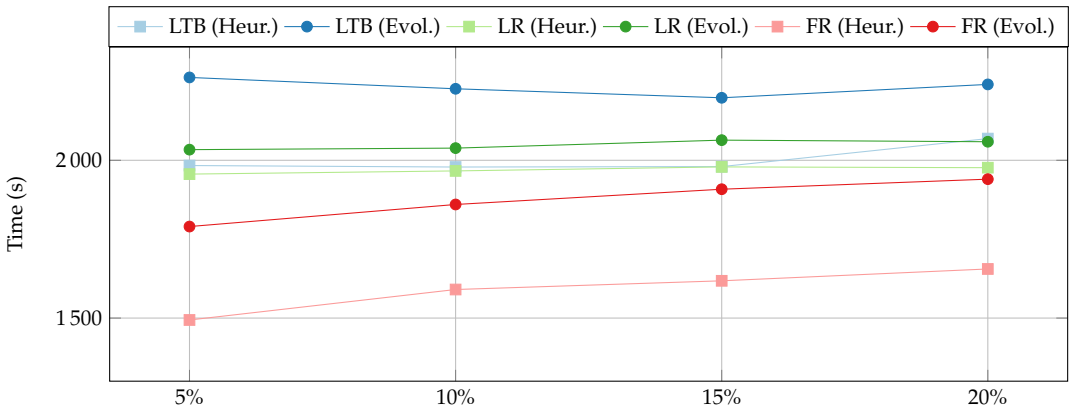


Figure 6.17. – Average runtimes for performing the WCET-aware traffic shaping optimization.

analysis are excluded. The full refill shaper implemented with the greedy heuristic has the consistently lowest average runtime, with a large gap to all other depicted approaches. It shows a slight increase on average runtime with a growing number of bus accesses per time interval to be reduced. Here, this is due to the larger number of performed iterations of the greedy heuristic as on average, more paths have to be adjusted. The full refill shaper implemented with the evolutionary algorithm shows the second lowest average runtime.

The lazy rate shaper implemented with the greedy heuristic shows a nearly constant runtime, mostly independent from the amount of bus accesses per time interval to be reduced. As the lazy rate shaper has a shaping profile of a step function with a step height of 1, a minimum distance between two accesses is already enforced for the lowest amount of bus accesses per time interval to be reduced. Therefore, the number of paths to be adjusted does not increase strongly with a larger number of bus accesses, but only the amount of delay to be inserted. The corresponding shaper implementation with the evolutionary-based approach also results in a mostly constant runtime, yet slightly higher on average.

The largest runtime is required on average when using the lazy token bucket shaper implemented with the evolutionary approach. This large optimization runtime on average is based on the more complex ILP model to solve for each profile adherence check. Whereas the full refill and lazy rate shaper adherence checks are implemented by simply deriving the value of $\eta^+(\Delta t)$, whereas Δt is set to period of the shaper, the adherence check for the lazy token bucket shaper extends the ILP formulation. This more complex ILP leads to longer average solving times for each profile adherence check.

6.5.4 Conclusion

Overall, a few different conclusions can be made concerning the effectiveness of WCET-aware traffic shaping for lowering the WCRT of tasks inside a multi-core system and the ideal shaper and implementation method to choose. In general, the full refill shaper offers the highest precision in terms of shaping a bus access profile with the least impact on a task's WCRT. While other shapers resulted in larger WCRT

reductions for the remaining tasks, this is due to an overly aggressive shaping which can be also reached, if required, using the full refill shaper by simply setting a tighter profile. Although the evolutionary-based implementation method showed slightly better results for the full refill shaper, the greedy heuristic has a significantly lower runtime on average, which would make it the more appropriate choice in most cases. When regarding a multi-core system with a fixed-priority-based bus arbitration, large WCRT reductions can be achieved by only shaping, e.g., the task allocated to the highest priority core. Yet, on average, the negative impact of shaping to the shaped task increases with the number of cores in the system. The average WCRT reductions inside a multi-core system with a round robin-based bus arbitration are quite smaller when compared to the fixed-priority bus systems, yet existing. With an increasing number of cores, the influence of shaping only the behavior of a single core further decreases. Therefore, when WCET-aware code-inherent traffic shaping is used for WCRT reduction in a round robin-based bus system, the number of shaped cores should be increased with an increasing number of cores if possible.

Overall, this potential use case showed that the presented techniques of code-inherent traffic shaping can be used to decrease the worst-case timings of tasks inside a multi-core system by reducing the overall bus contention. Especially in case one or more tasks meet their deadline with a slack, this slack can be used by traffic shaping. While this is one potential use case, the approach is not bound to this. Another possible use case is limiting the maximum number of messages sent between networks to avoid a buffer overflow as presented at the *International Conference on Embedded Software (EMSOFT)* in New York 2019 [OSF19].

7 Cooperative Combined Static Memory Allocation

The bus-aware memory allocation optimization presented in Chapter 4 demonstrated that bus-induced timing effects can have significant effects on the outcome of a WCET-directed optimization. Yet, due to the very detailed modeling of the potential bus state at every access, the optimization model does not scale very well for larger applications. Additionally, the optimization requires considerably strict assumptions on the bus architecture to achieve a higher predictability, such as the use of a TDMA arbitrated bus with a minimal slot size length. Though advocated for the use in hard real-time systems [CFG+10] and supported by popular on-chip buses such as, e.g., AMBA [MS06], most COTS multi-core architectures do not support a TDMA bus arbitration policy natively. As a possible solution, a TDMA arbitration may be enforced by software on architectures without TDMA support. Ziccardi et al. [ZCMV15] demonstrated such an approach to enforce a configurable TDMA bus arbitration on an AURIX TC277TU multi-core processor by expanding the real-time operating system ERIKA Enterprise [GBL+00]. Yet, this workaround requires the availability and the expansion of an operating system. Especially for programs with a significant ratio of bus access, this approach will further increase the WCET, as every access has to be handled using a specific operating system function.

In the previous chapter, the concept of WCET-aware code-inherent traffic shaping was introduced, and its potential use for improving the worst-case behavior of a multi-core real-time system was discussed. Simply speaking, the approach adapted the system's behavior by essentially "delaying" certain events of a program's event arrival function $\eta^+(\Delta t)$ to larger values of Δt. In the following section, a bus-aware memory allocation approach based on the analysis of event arrival functions is presented. Instead of delaying certain events of an event arrival functions as done using code-inherent traffic shaping, this optimization corresponds to "removing" certain steps of a program's event arrival function (as an access to a private memory does not generate a bus access).

The following memory allocation optimization tries to improve scalability by modeling the bus-related timing effects with a higher abstraction, yet still derived from a code-level. It no longer requires the strict restriction of a TDMA bus with minimal slot lengths, but can be applied to a variety of bus arbitration techniques. Here, a round robin-based bus arbitration (also known as fair) is assumed as it is a commonly used arbitration scheme for modern bus-based multi-core architectures. As it is a work-conserving bus arbitration policy, it has a high bus utilization. It also has tight upper bounds on the longest possible time until an access is granted, which prevents starvation and enables it for the use in hard real-time systems.

When using a work-conserving bus, the execution of a program on one core can easily influence the execution of another program on a different core. The memory allocation of a single program is therefore no longer isolated to only the program's

particular core, but rather influences the whole system. This constitutes the idea of a *cooperative* memory allocation. The optimization aims at finding a memory allocation for each program on each core which results in an overall schedulable system (meaning each program's WCRT is below its deadline), rather than optimizing the WCRT for each program separately. For example, although a program already provably meets its deadline, placing favorable additional parts to its private memory may decrease the WCRT of a concurrent program as it decreases the overall bus contention. This differs from simply optimizing the WCRT of the program in isolation, as the program parts to allocate to the private memory in order to improve the concurrent program's timing may easily not be on its very own worst-case execution path, or may even worsen its worst-case timing slightly (e.g., due to additional jump correction code).

While the memory allocation optimization presented in Chapter 4 only focused on placing the right code fragments into the cores' private memory, the optimization in this chapter takes code *and* data objects into account. This *combined* memory allocation decides which parts of the code and which data objects should remain in the shared memory and which should be allocated to a core's private memory in order to reach an overall schedulable system.

In favor of determinism, the optimization performs a *static* memory allocation, rather than a *dynamic* one. When applying a static memory allocation, program parts are moved to the corresponding memories during the startup phase of the system and remain there. In contrast, a dynamic memory allocation allows the memory contents to be replaced during the runtime of a program by re-allocating them. While this allows for a higher utilization of the memories (as, e.g., no longer needed parts can be replaced by other ones), it also increases the overhead as program parts have to be moved *during* the execution time. Although this overhead can be reduced by using dedicated techniques, such as, e.g., by initiating memory transfers ahead of time via DMA, it typically can not be removed fully.

Three different approaches are presented and evaluated in the following. The first approach is fully based on Integer Linear Programming. Here, the worst-case response time depending on the memory allocation is modeled as an ILP. The second technique is a greedy heuristic with auxiliary ILP models in intermediate steps. This approach precisely evaluates bus timing effects at each iteration and uses a simplified ILP model to allocate code. An approach based on evolutionary algorithms is introduced as a third option. The evolutionary-based approach serves as a generic baseline, as evolutionary algorithms are used in wide range of domains.

This chapter does not feature a dedicated related work section, as the related work to WCET-directed memory allocation was previously discussed in the context of the static instruction allocation for TDMA-based architectures in Section 4.1. Section 7.1 outlines the architectural assumptions made for the optimization. The fully ILP-based optimization is presented in Section 7.2. Here, Section 7.2.1 gives a short reintroduction of the bus-unaware base model, whereas Section 7.2.2 presents the actual bus-aware additions. Following, the greedy heuristic is presented in Section 7.3. The reference approach based on an evolutionary algorithm is shown in Section 7.4. Eventually, the different approaches are evaluated and their results are discussed in Section 7.5.

7.1 Architectural Assumptions

The assumed architecture of the system corresponds to the exemplary base architecture shown in Figure 2.6 (cf. Page 13) with N_c parallel homogeneous cores with private SPMs and shared memories which are connected to the bus. The bus arbitration is assumed to be round robin-based in the following. Yet, this assumption is exemplary, as it is not a necessary characteristic for the optimization approach. It is rather chosen as a predominant bus arbitration in modern bus-based multi-core architectures. E.g., a fixed priority-based bus arbitration could be used as well with adjustments. Also, a TDMA-based arbitration could be used, although this would remove the cooperative character of the optimization, as a memory allocation for one core would not influence the timing of another core in a TDMA arbitrated system.

It is assumed that one task is allocated per core. The task-to-core-mapping is assumed to be fixed, i.e., tasks do not migrate to other cores during runtime. The tasks' activation patterns are assumed to be arbitrary, as long as they can be expressed by event arrival functions $\eta^+(\Delta t)$.

7.2 ILP-based Combined Cooperative Allocation

In this section, an ILP formulation is presented which aims at lowering a task's WCRT below its deadline and tries to improve the timing of tasks on concurrent cores as well. Instead of describing the entire system with all tasks in one large ILP, each task is handled on its own. Using a single large ILP model for all tasks would significantly lower the approach's scalability. To actually benefit from a joint ILP model, a bus timing analysis would be required to be formulated into the ILP which would decrease the scalability even further. As done previously, lowercase Latin letters such as x are used as variables inside the ILP, whereas uppercase Latin letters such as C are used as constants.

The ILP-based approach is split into two steps: At first, a base ILP model for a combined memory allocation is created which is completely bus-unaware. This ILP model is solved and the resulting memory allocation is applied. In case the overall system becomes schedulable with this allocation, the optimization terminates as the goal of a schedulable system is reached. If not, the ILP model is extended to reflect bus contention. The initial solving of the bus-unaware model is required to gain an estimate of the overall bus contention in the system with some parts allocated to the private memories. As the final SPM allocation is not known, the exact level of bus contention (which depends on the memory allocation decisions of each task) is not known during the creation and solving of the ILP model for each task. The idea here is that the final SPM allocation will be somewhat similar to the bus-unaware allocation and hence the level of bus contention with this allocation applied is used as a reference for the bus-aware ILP model.

7.2.1 Bus-unaware Base Model

The ILP model will be explained using the exemplary control-flow graph shown in Figure 7.1. The blocks with a solid outline represent basic blocks, whereas the dashed circle represents a function call. If a basic block accesses a data object, the label of

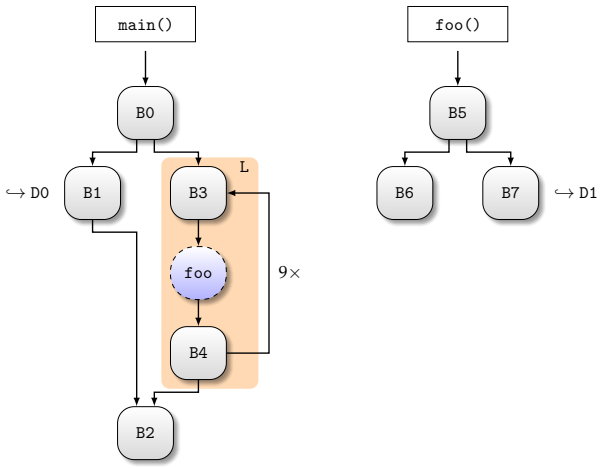


Figure 7.1. – Exemplary control-flow graph.

this data object is noted beside the basic block. As mentioned previously, a separate ILP model is created and solved for each individual task. For the sake of clarity, no task-specific indices are used for symbols unless otherwise noted. Each task-related symbol (e.g., a deadline D) is associated with the current task for which the ILP model is generated. The bus-unaware ILP's basic structure is a combined approach of the instruction SPM allocation presented by Falk and Kleinsorge [FK09] and the data SPM allocation presented by Suhendra et al. [SMRC05], whereas the instruction SPM allocation's ILP model is in turn based on the data SPM allocation's ILP model as well. The original data SPM allocation ILP model presented by Suhendra et al. is on the other hand a modified variant of the *Implicit Path Enumeration Technique*'s [LM95] dual problem.

As the ILP model used in the instruction SPM allocation for TDMA-arbitrated architectures presented in Chapter 4 is based on the ILP model by Falk and Kleinsorge as well, the following will include a brief recapitulation of previously introduced symbols. All timing-related constants are derived in isolation, i.e., no interferences from other cores are present. The essential additions and modifications which differentiate the base model in this chapter from the bus-unaware base model in Chapter 4 are the introduction of a timing gain for allocating data objects to the data SPM (cf. Equations (7.4) and (7.5)) and a constraint to limit the maximum number of data objects allocated to the data SPM (cf. Equation (7.15)). The following paragraphs will shortly re-introduce the base model with the aforementioned modifications, whereas the upcoming Section 7.2.2 introduces the actual cooperative memory allocation optimization.

For each basic block $\mathtt{B}$, an integer variable $w_{\mathtt{B}}$ is introduced, representing the maximum execution time when starting at basic block $\mathtt{B}$ until an exit of the function of which $\mathtt{B}$ is part of. Hence, if basic block $\mathtt{B}$ is the entrypoint of the program, $w_{\mathtt{B}}$ describes the WCET of the program. The worst-case execution time of a single execution of a basic block $\mathtt{B}$ is described by the constant $C^{+}_{\mathtt{B}}$. A constant $G_{\mathtt{B}}$ is introduced for every basic block, representing the gain in terms of execution cycles when basic block $\mathtt{B}$ is moved to the faster private memory. This can be derived by performing two subsequent WCET analyses. For the first analysis, all basic block are placed in the shared memory, whereas for the second analysis, all basic blocks are placed in the

(virtually enlarged) private memory. $G_{\mathtt{B}}$ can then be calculated as the difference in the worst-case execution time of a single execution of basic block B. Referring to basic block B0 of Figure 7.1, the corresponding variable $w_{\mathtt{B0}}$ is bounded below by the following two constraints:

$$w_{\mathtt{B0}} \geq C^{+}_{\mathtt{B0}} - x_{\mathtt{B0}} \cdot G_{\mathtt{B0}} + w_{\mathtt{B1}} + k_{\mathtt{B0,B1}} \tag{7.1}$$

$$w_{\mathtt{B0}} \geq C^{+}_{\mathtt{B0}} - x_{\mathtt{B0}} \cdot G_{\mathtt{B0}} + w_{\mathtt{L}} + k_{\mathtt{B0,B3}} \tag{7.2}$$

Equation (7.1) describes the path over the left-hand side in the CFG in Figure 7.1. The binary variable $x_{\mathtt{B0}}$ reflects the memory allocation decision of basic block B0. If set to 1, basic block B0 is allocated to the private memory and otherwise resides in the shared memory. $k_{\mathtt{B0,B1}}$ denotes the additional amount of cycles due to the execution of additional jump correction code in case basic block B0 and its successor B1 are not placed in the same memory. This is the case when, e.g., a previous relative jump cannot cover the address offset between the two memory regions and hence has to be replaced by a more costly indirect jump. The variable $k_{\mathtt{B0,B1}}$ is set using the following equation:

$$k_{\mathtt{B0,B1}} = K_S \cdot (\overline{x}_{\mathtt{B0}} \wedge x_{\mathtt{B1}}) + K_P \cdot (x_{\mathtt{B0}} \wedge \overline{x}_{\mathtt{B1}}) \tag{7.3}$$

K_S is the approximated number cycles for executing the additional jump correction code for this jump from the shared memory. K_P is the counterpart if the jump originates in the private memory. This differentiation is done, as the required execution time to execute the additional jump correction code is depending from which memory these instructions are fetched.

The variable $w_{\mathtt{B1}}$ referenced in Equation (7.1) is bounded by the following constraint accordingly:

$$w_{\mathtt{B1}} \geq C^{+}_{\mathtt{B1}} - x_{\mathtt{B1}} \cdot G_{\mathtt{B1}} - x_{\mathtt{D0}} \cdot G^{\mathtt{B1}}_{\mathtt{D0}} + w_{\mathtt{B2}} + k_{\mathtt{B1,B2}} \tag{7.4}$$

For the most parts, Equation (7.4) follows the structure as the previous constraints presented for basic block B0. Yet, since it accesses the data object D0, the additional gain term $x_{\mathtt{D0}} \cdot G^{\mathtt{B1}}_{\mathtt{D0}}$ for this data object is added. The binary variable $x_{\mathtt{D0}}$ represents the allocation decision for the data object D0.

A gain constant $G^{\mathtt{B}}_{\mathtt{D}}$ is introduced for each data object D accessed in basic block B. $G^{\mathtt{B}}_{\mathtt{D}}$ denotes by how many cycles the worst-case execution time of basic block B is reduced when data object D is placed in the core's private memory. This gain is derived by the following equation:

$$G^{\mathtt{B}}_{\mathtt{D}} = A^{\mathtt{D}}_{\mathtt{D,B}} \cdot (F_{\mathtt{S}} - F_{\mathtt{P}}) \tag{7.5}$$

$A^{\mathtt{D}}_{\mathtt{D,B}}$ denotes the number of times data object D is accessed during a single execution of basic block B. The access latency of the shared memory is described by $F_{\mathtt{S}}$, whereas $F_{\mathtt{P}}$ describes the access latency of the private memory.

Equation (7.2) describes the execution path over the right-hand side of the CFG in Figure 7.1. Loops are represented by meta blocks in the model. The tail-controlled loop in the shown control-flow graph is represented using the meta block L with its

loop members B3 and B4. The execution time of the loop block L and its loop members is described using the following constraints:

$$w_{\mathtt{L}} \geq 10 \cdot w_{\mathtt{B3}} + w_{\mathtt{B2}} + k_{\mathtt{B4},\mathtt{B2}} + 9 \cdot k_{\mathtt{B4},\mathtt{B3}} \tag{7.6}$$

$$w_{\mathtt{B3}} \geq C^{+}_{\mathtt{B3}} - x_{\mathtt{B3}} \cdot G_{\mathtt{B3}} + w_{\mathtt{B5}} + k_{\mathtt{B3},\mathtt{B5}} + w_{\mathtt{B4}} \tag{7.7}$$

$$w_{\mathtt{B4}} \geq C^{+}_{\mathtt{B4}} - x_{\mathtt{B4}} \cdot G_{\mathtt{B4}} \tag{7.8}$$

Equation (7.6) bounds the worst-case execution time when starting at the loop's entrance B3 up to the program's termination. As the back-edge of the loop is executed 9 times, the loop body is executed 10 times, hence the $10 \cdot w_{\mathtt{B3}}$ term. $w_{\mathtt{B2}}$ is added as the loop's succeeding basic block. The function call at the end of basic block B3 is modeled by adding the worst-case execution time of the called function foo to $w_{\mathtt{B3}}$. Here, the variable $w_{\mathtt{B5}}$ represents the WCET of function foo, as it is the function's entry basic block. Note that only for the calling edge $\mathtt{B3} \rightarrow \mathtt{B5}$, the costs of a potential jump correction code are added, but not for the return, as in most architectures, a return is already performed by an indirect jump (typically by moving the contents of a so-called *link register* into the program counter).

The remaining basic block B2 of function main is bounded using the following constraint:

$$w_{\mathtt{B2}} \geq C^{+}_{\mathtt{B2}} - x_{\mathtt{B2}} \cdot G_{\mathtt{B2}} \tag{7.9}$$

The three basic blocks of function foo are handled accordingly:

$$w_{\mathtt{B5}} \geq C^{+}_{\mathtt{B5}} - x_{\mathtt{B5}} \cdot G_{\mathtt{B5}} + w_{\mathtt{B6}} + k_{\mathtt{B5},\mathtt{B6}} \tag{7.10}$$

$$w_{\mathtt{B5}} \geq C^{+}_{\mathtt{B5}} - x_{\mathtt{B5}} \cdot G_{\mathtt{B5}} + w_{\mathtt{B7}} + k_{\mathtt{B5},\mathtt{B7}} \tag{7.11}$$

$$w_{\mathtt{B6}} \geq C^{+}_{\mathtt{B6}} - x_{\mathtt{B6}} \cdot G_{\mathtt{B6}} \tag{7.12}$$

$$w_{\mathtt{B7}} \geq C^{+}_{\mathtt{B7}} - x_{\mathtt{B7}} \cdot G_{\mathtt{B7}} - x_{\mathtt{D1}} \cdot G^{\mathtt{B7}}_{\mathtt{D1}} \tag{7.13}$$

This set of constraints models the control-flow graph, as well as a lower bound on the WCET. Additionally, constraints limiting the total number of allocated objects to the private memories are introduced:

$$\begin{aligned} S_{\mathrm{ISPM}} \geq\ & S_{\mathtt{B0}} \cdot x_{\mathtt{B0}} + Q_{\mathtt{B}} \cdot (x_{\mathtt{B0}} \wedge \overline{x}_{\mathtt{B1}}) + Q_{\mathtt{B}} \cdot (x_{\mathtt{B0}} \wedge \overline{x}_{\mathtt{B3}}) \\ & + S_{\mathtt{B1}} \cdot x_{\mathtt{B1}} + Q_{\mathtt{B}} \cdot (x_{\mathtt{B1}} \wedge \overline{x}_{\mathtt{B2}}) \\ & + S_{\mathtt{B2}} \cdot x_{\mathtt{B2}} \\ & + S_{\mathtt{B3}} \cdot x_{\mathtt{B3}} + Q_{\mathtt{B}} \cdot (x_{\mathtt{B3}} \wedge \overline{x}_{\mathtt{B5}}) \\ & + \ldots \\ & + S_{\mathtt{B7}} \cdot x_{\mathtt{B7}} \end{aligned} \tag{7.14}$$

$$S_{\mathrm{DSPM}} \geq S_{\mathtt{D0}} \cdot x_{\mathtt{D0}} + S_{\mathtt{D1}} \cdot x_{\mathtt{D1}} \tag{7.15}$$

$Q_{\mathtt{B}}$ is the number of additional bytes due to jump correction code. In case a basic block resides in the private memory and its successor not, $Q_{\mathtt{B}}$ bytes have to be inserted into the private memory for the jump correction code. $S_{\mathtt{B}}$ denotes the size of basic block B in bytes, whereas $S_{\mathtt{D}}$ is the counterpart for the data object D. The size of the private

instruction (resp. data) memory is S_{ISPM} (resp. S_{DSPM}). Furthermore, the optimization objective is introduced into the ILP model:

$$\min : w_{\text{B0}} \tag{7.16}$$

When solving this ILP model, an optimal allocation of basic blocks and data objects is found in terms of the model to minimize the program's WCET. This is done for every task on each core, and the basic blocks and data objects are allocated according to the ILP solutions. Subsequently, a timing analysis (including bus contention) for the whole system is carried out to check, whether all tasks already meet their deadlines or not. If all tasks already meet their deadlines, the optimization can terminate at this point, as the overall goal of a schedulable system is already reached. In case the system is not schedulable yet, the ILP model is extended to include bus contention and a cooperative part.

7.2.2 Bus-aware Part

As done in the previous section, also the bus-aware part is introduced using the exemplary control-flow graph shown in Figure 7.1. The bus-aware extensions to the ILP model cover two main aspects: 1) Bus-related timings due to bus contention are reflected within the ILP model. 2) A cooperative allocation is performed, such that basic blocks or data objects are chosen which may help to decrease the WCRT of concurrent cores. The main idea of the bus-aware ILP model is to perform a "quasi hierarchical" optimization: The main focus lies on finding a memory allocation which leads to the corresponding task meeting its deadline. If such an allocation can be found, those basic blocks and data objects which potentially cause the most bus contention are allocated to the private memories as well in order to decrease the WCRT of concurrent tasks.

The previously found memory allocation by solving the bus-unaware ILP is used to derive an estimate on the bus contention. More precisely, the accumulated worst-case bus blocking time for each task is derived with the memory allocation applied and used as a reference. The accumulated worst-case bus blocking time L_i of a task τ_i describes the maximum amount of cycles which are spent due to bus stalling during a single execution of task τ_i. This obviously depends on the maximum number of bus accesses performed by task τ_i, as well as the maximum number of concurrent bus accesses performed by other cores. As this in turn is depending on the memory allocation of basic blocks and data objects and since the final allocation is unknown, L_i is determined with the previously found allocation applied as an estimate. When using a round robin arbitrated bus, L_i can be determined using the following equation [JHH15]:

$$L_i = \sum_{j \in \Gamma, j \neq i} \min \left(A^+_{\text{E},i},\ \eta^+_j \left(R^+_i \right) \right) \cdot F_{\text{S}} \tag{7.17}$$

The set Γ contains all tasks of the system. $A^+_{\text{E},i}$ is the maximum number of bus accesses performed by task τ_i during one complete execution. $\eta^+_j \left(R^+_i \right)$ describes the maximum number of bus accesses performed by task τ_j during a time interval equal to the length of the worst-case response time of task τ_i. The net access latency of the shared memory

is denoted by F_S. The idea behind Equation (7.17) is as follows: In a round robin arbitrated bus, the bus access priority of each core is rotated after each performed bus access. Therefore, a single access inside a system of N_C cores can be delayed by at most $(N_C - 1) \cdot F_S$ cycles [BRS11] in any case. More precisely, each access of τ_i may be delayed up to F_S cycles per potentially concurrent access per core. As it is assumed that one task is allocated to each core, the maximum number of accesses per core corresponds to the maximum number of accesses of the allocated task. The min-term in Equation (7.17) evaluates the maximum number of competing accesses during one execution of task τ_i.

Example 7.1. *A system with $N_C = 4$ cores is assumed and one task per core. Task τ_0 is allocated to core 0, τ_1 to core 1, etc. The system features a shared memory which is accessed via a round robin arbitrated bus. The WCRT of task τ_0 is $R_0^+ = 1\,000$ cycles and it performs at most $A_{E,0}^+ = 10$ bus accesses during one complete execution. The maximum number of accesses performed by the other tasks during this time interval length are as follows: $\eta_1^+(1\,000) = 5$, $\eta_2^+(1\,000) = 18$ and $\eta_3^+(1\,000) = 0$. As task τ_1 only performs up to 5 accesses during a complete execution of task τ_0, only 5 of task τ_0's 10 accesses may be competing with task τ_1. Task τ_2 performs up to 18 accesses during the execution time of τ_0, which means that potentially all of τ_0's accesses may be competing with accesses of τ_2. Finally, as τ_3 initiates 0 accesses during the execution of τ_0, no accesses of τ_0 are potentially competing with τ_3. This relates to an accumulated worst-case blocking time of $(5 + 10 + 0) \cdot F_S$ for task τ_0, which relates to the given Equation* (7.17).

The actual value of R_i^+ is derived using the timing analysis methods described in Section 6.5.3. This timing analysis also derives the required upper event arrival functions $\eta_j^+\left(R_i^+\right)$ of each concurrent task j using the methods presented in Chapter 5. The accumulated worst-case bus blocking time L_i is derived for every task τ_i with the memory allocation applied determined by the bus-unaware ILP model from Section 7.2.1. L_i will be used in the bus-aware ILP model to tighten the estimated WCRT inside the model. After deriving this estimate for the ILP, the temporary memory allocation found using the bus-unaware base model is undone and all data objects and basic blocks are re-assigned to their default allocation (the shared memory). Besides, all potentially added jump correction code is removed as well, such that all programs are again in their original state as before the optimization.

As the reasoning behind the bus-aware ILP model is not to simply minimize the WCRT of each task, but rather to decrease a task's WCRT below its deadline and then to try to reduce the overall bus contention, the original objective of Equation (7.16) needs to be replaced. Therefore, the objective is replaced by the following term, where λ is an integer variable describing by how many cycles the current task is violating its deadline, and ϕ is an integer variable used as a figure of merit describing the bus contention caused by the task:

$$\min : \lambda + \phi \tag{7.18}$$

The deadline violation λ is defined using the task's worst-case response time r and its deadline D:

$$\lambda \geq r - D \tag{7.19}$$

$$\lambda \geq 0 \tag{7.20}$$

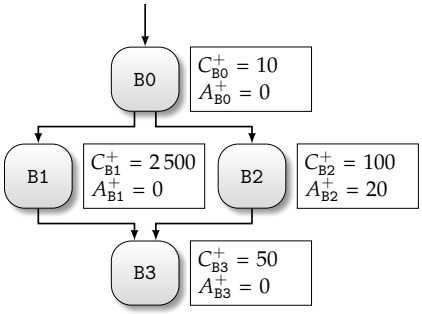


Figure 7.2. – Another exemplary control-flow graph.

As there is no need to decrease the task's WCRT below its deadline, the deadline violation variable λ is given a lower bound of 0. While w_{B0} represents the WCET (without any interference) of the task inside the bus-unaware ILP, the bus-aware model requires the inclusion of bus-related timings, represented by the task's worst-case response time r. Assuming a timing anomaly-free architecture, a conservative WCRT is described by $w_{B0} + L$, as the task cannot be blocked longer than L cycles due to competing bus accesses. Yet, this is only valid as long as the maximum blocking time L with the final memory allocation applied is lower than or equal to the maximum blocking time L used within the ILP description (derived using the temporary memory allocation of the bus-unaware ILP model). Since the final allocation is unknown, the maximum blocking time L is only an estimate here. It is assumed that for most cases, the maximum blocking time with the final allocation applied will be lower than the blocking time with the temporary allocation. The aim of the bus-aware cooperative ILP model is not to simply minimize each task's WCRT, but rather to reduce a task's WCRT as much as needed and then further allocate program parts which cause the most bus accesses to a private memory. Therefore, it is assumed that the final allocation will have less overall bus accesses compared to the temporary allocation (derived using the bus-unaware ILP), since the bus-unaware ILP model only aims at minimizing each WCET, the bus-aware ILP model also tries to explicitly reduce the overall number of bus accesses if possible.

Yet, this WCRT estimate may be significantly pessimistic, as it assumes all of the bus accesses A^+_E potentially performed during one execution are part of the worst-case execution path, since L is derived on a system-level. This pessimism is illustrated in Figure 7.2 and discussed in the following example.

Example 7.2. *The left-hand side of the control-flow graph of Figure 7.2 features no bus accesses, only local computation. In contrast, the right hand-side features several bus accesses, but a rather low computational demand. The conservative WCRT estimation would now correspond to* $w_{B0} = 2\,560 + L$ *cycles, although no bus accesses are performed along the left-hand side path which is the WCEP in an isolated case.*

At the same time, assuming the worst-case delay π *for each bus access results in a safe estimation of the task's WCRT as well. In case of a round robin bus arbitration, the worst-case delay of a single bus access is, as previously mentioned, described by* $\pi = (N_C - 1) \cdot F_S$. *As both approaches (adding* L *to the task's WCET or assuming the worst-case latency for*

each access) lead to a safe WCRT, the lower one of both can be chosen. Regarding the small exemplary CFG shown in Figure 7.2, this would correspond to the following term:

$$\min(2\,560 + L,\ 160 + 20 \cdot \pi) \tag{7.21}$$

As can be seen, it depends on the task itself and on the overall system, which of the two estimates will be tighter.

The principle shown in the previous example is applied in the bus-aware ILP model. Returning to the originally discussed control-flow graph depicted in Figure 7.1 and its ILP model, the task's WCRT r is as follows:

$$r = w_{\mathtt{B0}} + L \cdot g \tag{7.22}$$

The binary ILP variable g describes whether each bus access is assumed with its worst-case delay ($g = 0$), hence already included in $w_{\mathtt{B0}}$, or whether the maximum accumulated bus blocking time L should be added. As the ILP's objective is set to minimize, the ILP solver will choose the lower of both options.

Since the deadline violation variable λ has a lower bound of 0, the ILP's objective value does not decrease as soon as a memory allocation leads to a schedulable task. Therefore, the second term in the objective function ϕ, which represents the bus contention caused by the current task, is subject to be minimized. Though it would be possible to describe the maximum number of bus accesses performed during one complete task execution inside the ILP (basically by inserting the ILP model described in Chapter 5 alongside), it would heavily increase the ILP's complexity and is therefore practically not feasible. Thus, the bus contention ϕ is estimated as a weighted sum of all potential bus accesses:

$$\begin{aligned}
\phi = {} & E_{\mathtt{B0}} \cdot A^{\mathrm{I}}_{\mathtt{B0}} \cdot (\overline{x}_{\mathtt{B0}} \vee \overline{y}) && (7.23)\\
& + Q_{\mathrm{A}} \cdot E_{\mathtt{B0}} \cdot ((\overline{x}_{\mathtt{B0}} \wedge x_{\mathtt{B1}}) \vee \overline{y}) + Q_{\mathrm{A}} \cdot E_{\mathtt{B0}} \cdot ((\overline{x}_{\mathtt{B0}} \wedge x_{\mathtt{B3}}) \vee \overline{y}) && (7.24)\\
& + E_{\mathtt{B1}} \cdot A^{\mathrm{I}}_{\mathtt{B1}} \cdot (\overline{x}_{\mathtt{B1}} \vee \overline{y}) + E_{\mathtt{B1}} \cdot A^{\mathrm{D}}_{\mathtt{D0},\mathtt{B1}} \cdot (\overline{x}_{\mathtt{D0}} \vee \overline{y}) && (7.25)\\
& + Q_{\mathrm{A}} \cdot E_{\mathtt{B1}} \cdot ((\overline{x}_{\mathtt{B1}} \wedge x_{\mathtt{B2}}) \vee \overline{y}) && (7.26)\\
& + \ldots && (7.27)
\end{aligned}$$

Where y is a binary ILP variable, representing whether the task's estimated WCRT is below its deadline ($y = 1$) or not ($y = 0$). It is defined as follows:

$$y = \begin{cases} 1 & \text{if } \lambda = 0, \\ 0 & \text{else.} \end{cases} \tag{7.28}$$

The overall idea of the variable ϕ is to reflect the potential bus contention caused by the current task. Bus accesses can be caused by either fetching code or reading or writing of a data object from the shared memory. $A^{\mathrm{I}}_{\mathtt{B}}$ describes the maximum number of bus accesses required to fetch all instructions from basic block `B`. Similarly, $A^{\mathrm{D}}_{\mathtt{D},\mathtt{B}}$ describes the number of times the data object `D` is accessed during a single execution of basic block `B`. To get an estimate how many bus accesses each basic block or data object could cause accumulated, these constants are multiplied by the maximum number of executions $E_{\mathtt{B}}$ of the corresponding basic block `B`. Note that $E_{\mathtt{B}}$ describes the maximum

number of executions of basic block B possible during one complete task execution, independent from the task's WCEP or similar. Referring to the control-flow graph in Figure 7.1, these maximum number of executions of the basic blocks can be easily seen (e.g, $E_{\mathtt{B0}} = E_{\mathtt{B1}} = E_{\mathtt{B2}} = 1$, $E_{\mathtt{B3}} = ... = E_{\mathtt{B7}} = 10$). The maximum execution count E does not refer to a single execution path, but to all execution paths possible. Thus, $E_{\mathtt{B1}} = 1$ and $E_{\mathtt{B3}} = 10$, although basic blocks B1 and B3 are mutually exclusive. The idea is similar to an event arrival function $\eta^+(\Delta t)$ describing all possible sub-paths, so to speak a superimposition of all sub-paths. For more complex control-flow graphs, $E_{\mathtt{B}}$ can be derived by, e.g., a modified IPET model (see Appendix D for an example). Therefore, the term $E_{\mathtt{B0}} \cdot A^{\mathrm{I}}_{\mathtt{B0}}$ in Equation (7.23) corresponds to the maximum number of bus accesses due to fetching the instructions of basic block B0 from the shared memory during one complete task execution. Yet, this term is only added if the task does not meet its deadline yet ($y = 0$) or if the basic block is not placed in the private memory ($x_{\mathtt{B0}} = 0$). Since every additive term is added as long as the task's WCRT is above its deadline, ϕ remains at its maximum value possible. This enforces a hierarchical optimization: As ϕ is forced to its largest possible value as long as the task violates its deadline, the objective value (cf. Equation (7.18)) is only decreased by lowering the task's WCRT. If the task's WCRT is equal to or below its deadline ($\lambda = 0$, $y = 1$), ϕ is no longer forced to its largest value, enabling a lower objective value by placing the basic blocks or data objects into the private memory which potentially cause the highest number of bus accesses. In case a basic block is allocated to the private memory (and the task's deadline is not violated), it no longer needs to be fetched from the shared memory, hence the additive term for this block is forced to 0 (cf. Equation (7.23)).

Yet, as placing two succeeding basic blocks into different memories may cause additional code to be executed due to jump correction code, this also may lead to additional instructions required to be fetched from the shared memory. This is reflected by Equation (7.24). The constant Q_{A} estimates the number of bus accesses required to fetch the additional jump correction code from the shared memory. As the jump correction code is added to the memory region of the jump's origin, this constant is only added in case the jump's source basic block is placed in the shared memory ($x_{\mathtt{B0}} = 0$) and the target block is placed in the private memory ($x_{\mathtt{B1}} = 1$ or $x_{\mathtt{B3}} = 1$). The term $E_{\mathtt{B1}} \cdot A^{\mathrm{D}}_{\mathtt{D0},\mathtt{B1}}$ in Equation (7.25) represents the maximum accumulated number of accesses to the data object D0 in basic block B1 during a complete task execution. If data object D0 is placed into the private memory (and the task's deadline is not violated), this term evaluates to 0 and ϕ is reduced.

Overall, the "quasi secondary" objective ϕ expresses the *cooperative* character of the memory allocation optimization. As soon as the task's deadline is not violated anymore, it is tried to reduce the maximum number of bus accesses caused by the task to potentially lower the WCRT of concurrent tasks. Moreover, the WCRT of the task under optimization may even be increased to reduce the overall bus contention as long as the task's deadline is not violated.

As described by Equation (7.22), the worst-case response time of the task is set by either adding the worst-case accumulated bus blocking time L to the value the isolated WCET of the task, or by assuming the worst-case delay for each access. The first case is represented by $g = 1$, whereas the latter is represented by $g = 0$. A new variable $\beta_{\mathtt{B}}$ is added for every basic block B, representing the worst-case bus access

delay when executing basic block B and $g = 0$. This variable is added as an additive term to every constraint bounding the accumulate worst-case execution time $w_{\mathtt{B}}$ of block B. Basic block B1 from Figure 7.1 is chosen as an example to showcase this. The previous Equation (7.4) (cf. Page 167) is replaced by the following constraint:

$$w_{\mathtt{B1}} \geq C^{+}_{\mathtt{B1}} - x_{\mathtt{B1}} \cdot G_{\mathtt{B1}} - x_{\mathtt{D0}} \cdot G^{\mathtt{B1}}_{\mathtt{D0}} + w_{\mathtt{B2}} + k_{\mathtt{B1},\mathtt{B2}} + \beta_{\mathtt{B1}} \tag{7.29}$$

$\beta_{\mathtt{B}}$ is defined by the maximum number of bus accesses performed by basic block B (depending on the memory allocation) which is multiplied by the worst-case access delay π of a single access:

$$\beta_{\mathtt{B1}} = A^{\mathrm{I}}_{\mathtt{B1}} \cdot \pi \cdot (\overline{g} \wedge \overline{x}_{\mathtt{B1}}) + A^{\mathrm{D}}_{\mathtt{D0},\mathtt{B1}} \cdot \pi \cdot (\overline{g} \wedge \overline{x}_{\mathtt{D0}}) + Q_{\mathrm{A}} \cdot \pi \cdot (\overline{g} \wedge \overline{x}_{\mathtt{B1}} \wedge x_{\mathtt{B2}}) \tag{7.30}$$

$A^{\mathrm{I}}_{\mathtt{B1}}$ denotes the maximum number of bus accesses required to fetch all instructions from basic block B1, such that $A^{\mathrm{I}}_{\mathtt{B1}} \cdot \pi$ corresponds to the worst-case delay due to bus contention for fetching B1 from the shared memory. This term is only added if each access is assumed with its worst-case bus delay ($g = 0$) and the basic block is not assigned to the private memory ($x_{\mathtt{B1}} = 0$). Similarly, the term $A^{\mathrm{D}}_{\mathtt{D0},\mathtt{B1}} \cdot \pi$ denotes the bus access delay due to accesses to the data object D0 during the execution of B1, in case each bus access is assumed with the worst-case. In case basic block B1 resides in the shared memory, yet its successor B2 is placed in the private memory, Q_{A} additional bus accesses are introduced due to the jump correction code. This is reflected by the last term in Equation (7.30).

Finally, when solving the ILP per core, it is tried to find a memory allocation which leads to a WCRT below or equal to the task's deadline in the first place, and which tries to reduce the overall bus contention as much as possible in the second.

7.3 Greedy Heuristic

In the following, an alternative approach to the previously presented purely ILP-based optimization is discussed. The goal of this greedy heuristic is the same, to find a static memory allocation for all basic blocks and data objects, such that all tasks will provably meet their deadlines in any case. The previous approach has two potential drawbacks:

1. The size of the ILP may quickly increase compared to the bus-unaware base model due to Equations (7.23) to (7.26), as well as Equation (7.30) which needs to be added for each basic block. This may restrict the scalability of the approach.

2. The worst-case accumulated bus blocking time L used within the ILP model is only an estimate, derived from the memory allocation which resulted from the bus-unaware model. In case the memory allocation found by the bus-aware model differs significantly from the one determined by the bus-unaware model, L might be too imprecise. This could lead to a wrong conception of the worst-case execution path inside the ILP model and would result in a sub-optimal memory allocation.

The key idea of the greedy heuristic presented in this section is to iteratively check, whether the current memory allocation leads to all tasks meeting the deadlines or

Algorithm 7.1 Greedy Heuristic for a Combined Cooperative Static Memory Allocation.

Inputs: Γ – Set with all tasks; $\eta_{\Gamma}^{+}(\Delta t)$ – Upper event arrival functions for all tasks in the system; N_{M} – Maximum number of basic blocks or data objects allocated to each SPM per iteration

Output: –

```
1:  z ← true
2:  while z do
3:      z ← false
4:      s ← performTimingAnalysis( Γ, η⁺_Γ(Δt))
5:      if s then
6:          break
7:      end if
8:      for τ_n ∈ Γ do
9:          K_n ← getBestBB( τ_n, Γ, N_M )
10:         O_n ← getBestData( τ_n, Γ, N_M )
11:         allocate( K_n ∪ O_n )
12:         if K_n ≠ ∅ then
13:             fixCFG( τ_n )
14:         end if
15:         if K_n ∪ O_n ≠ ∅ then
16:             z ← true
17:         end if
18:     end for
19: end while
```

not. In case a task does not meet its deadline yet, it is estimated from which program parts to be allocated to the private memory this task would benefit the most. A fixed number of these parts is then allocated to the private memory. In case a task does already meet its deadline, while others in the system do not yet, allocating additional parts of the task not violating its deadline might improve the timing of the other tasks. A fixed number of program parts potentially helping the most to other tasks is then allocated to the private memory.

The general principle of the greedy heuristic is shown in Algorithm 7.1. The variable z is used as a flag, indicating whether the memory allocation changed during the last iteration (z = true) or not (z = false). In case the memory allocation was not changed anymore for any task in the system, the main loop spanning from Line 2 to 19 is exited. At the start of each iteration, z is set to false and a timing analysis of the system is performed in Line 4. The timing analysis derives the worst-case response time for each task with the current memory allocation applied and returns a Boolean value s which is true in case all tasks meet their deadline and otherwise false. In case all tasks meet their deadline with the current memory allocation, the algorithm is stopped by exiting the main loop in Line 6. Otherwise, an inner loop spanning over Lines 8 to 18 iterates over all tasks, where the set Γ contains all tasks of the system.

In Line 9, the most promising N_{M} basic blocks of task τ_n to improve the system's timing behavior are determined and stored in the set $\mathcal{K}_n$. Similarly, the N_{M} best data objects of task τ_n are chosen in Line 10 and stored in the set $\mathcal{O}_n$. Both functions may

also return less than N_M elements, in case the corresponding private memory cannot fit more, yet never more than N_M. N_M is a user-definable constant, representing a trade-off between accuracy and required algorithm runtime. The larger N_M is chosen, the more allocations are done per iteration, hence less iterations and therefore costly timing analyses are done in total. On the downside, a larger N_M constant may lead to non-ideal allocations, as, e.g., the worst-case execution path of a program already switched, yet this is only uncovered in the next timing analysis. Here, the constant N_M is used for both, basic blocks and data objects, as it should be a simple parameter to adjust the call frequency of the timing analysis. The actual implementations of these two functions in Lines 9 and 10 are discussed in the upcoming paragraphs. Subsequently, the determined basic blocks and data objects are allocated to the private memory. To avoid loops of un- and re-assigning objects to the private memory, objects are only added to the private memory and are not unassigned again. In case new basic blocks were allocated to the private memory, potentially broken jumps are fixed in Line 13. If new parts of task τ_n (basic blocks or data objects) were allocated to the private memory in this iteration, z is set to true in Line 16.

The algorithm is executed until either all tasks meet their deadlines (as this is the overall goal), or no new allocation has been done in the entire system. In the following, the two key functions *getBestBB()* and *getBestData()* are discussed.

getBestBB() This function aims at returning the N_M most promising basic blocks of task τ_n to improve the overall system's worst-case timing behavior. In case τ_n violates its deadline, the function chooses basic blocks to decrease the WCRT of the task itself. Otherwise, it tries to select the basic blocks which are the most likely to decrease the timing of other tasks which do not meet their deadline yet. As *getBestBB()* also takes into consideration the leftover instruction SPM size in the current iteration, it may return less than N_M basic blocks if the SPM does not offer any more space. Overall, the problem to be solved by this function can be seen as a complex Knapsack problem [SDK75], in which every decision to choose an element will potentially alter the cost and benefit of the other elements (e.g., due to the additional jump correction). To get a satisfactory estimation which also includes predictions on the decisions' side effects, the function *getBestBB()* is implemented using an ILP. Event though at a first glance, it seems counterintuitive to use an ILP to solve the potential scaling problems of the ILP-based approach presented in the previous section, it is important to note that the ILP used in the *getBestBB()* function is magnitudes smaller compared to the previously introduced ILP model. The ILP used in this function only describes the problem of which basic blocks to choose to get the greatest potential timing gain, but it does not model the control-flow graph or any bus-effects.

The objective function of the ILP is set to maximize the achievable timing gain by selecting up to N_M basic blocks, while not exceeding the leftover private memory. In each iteration of Algorithm 7.1, a separate ILP is generated and solved for each core. If not further specified, each task-related variable or constant belongs to the task for which the ILP is being solved. The set of all basic blocks currently allocated to the shared memory is denoted as $\mathcal{G}_B$, whereas the set $\overline{\mathcal{G}_B}$ contains all basic blocks which are assigned to the private memory. These sets can be directly deduced by the *getBestBB()* function by iterating over all basic blocks of the given task τ_n and evaluating whether a basic block is currently assigned to the private SPM or the shared

memory. W_{ISPM} is the number of bytes already occupied in the private instruction memory in the current iteration of the heuristic. The basic set of constraints which limit the total number of basic blocks being chosen are implemented as follows:

$$N_{\text{M}} \geq \sum_{i \in \mathcal{G}_{\text{B}}} x_i \tag{7.31}$$

$$\begin{aligned} S_{\text{ISPM}} - W_{\text{ISPM}} \geq & \sum_{i \in \mathcal{G}_{\text{B}}} \left(x_i \cdot S_i + \sum_{j \in (\mathcal{P}_i \cap \mathcal{G}_{\text{B}})} Q_{\text{B}} \cdot (\overline{x}_i \wedge x_j) - \sum_{j \in (\mathcal{P}_i \cap \overline{\mathcal{G}_{\text{B}}})} Q_{\text{B}} \cdot x_j \right) \\ & - \sum_{i \in \overline{\mathcal{G}_{\text{B}}}} \sum_{j \in (\mathcal{P}_i \cap \mathcal{G}_{\text{B}})} Q_{\text{B}} \cdot x_j \end{aligned} \tag{7.32}$$

Equation (7.31) ensures that the total number of newly chosen basic blocks is lower than or equal to the user-defined constant N_{M}. Equation (7.32) handles the size constraint of the private instruction memory.

The left-hand side of Equation (7.32) describes the remaining bytes in the I-SPM at this iteration step. The first two additive terms on the right-hand side of Equation (7.32) describe the additional number of bytes due to the newly allocated basic blocks, as well as the additional space required for jump correction code. The term $Q_{\text{B}} \cdot (\overline{x}_i \wedge x_j)$, where j is a predecessor of i, describes the additional bytes due to jump correction code in case basic block i remains in the shared memory and j is placed in the private memory. Note that both basic blocks, i and j, are not allocated to the private memory at this iteration. Therefore only "new" jump correction code is described by this term, as W_{ISPM} already includes the additional jump correction code added from previous *fixCFG()* calls. As previously inserted jump correction code may become obsolete with new basic blocks being allocated to the private memory, the first subtractive term of Equation (7.32) accounts for jump correction code to be removed from the private memory. Note that the predecessor j is already allocated to the private memory here, whereas i is not. Similarly, the last subtractive term of Equation (7.32) handles the reduction of basic block j's size in case it is allocated to the private memory in this iteration, yet its successor i was already assigned to the private memory during a previous iteration. Therefore, j's size S_j was already increased in a previous iteration to include jump correction code. This additional jump correction code is no longer needed, if in this iteration also basic block j is assigned to the private memory.

The timing gain to be maximized in this ILP is a sum of the task's estimated *own* gain μ and the *cooperative* gain ν. The task's own gain μ represents the estimated decrease in worst-case response time, ignoring the potential positive influences on the concurrent tasks. It is defined using the following constraint:

$$\begin{aligned} \mu = & \sum_{i \in \mathcal{G}_{\text{B}}} \left(x_i \cdot G_i \cdot X_i - \sum_{j \in (\mathcal{P}_i \cap \mathcal{G}_{\text{B}})} k_{j,i} \cdot X_{j,i} + \sum_{j \in (\mathcal{P}_i \cap \overline{\mathcal{G}_{\text{B}}})} K_P \cdot X_{j,i} \cdot x_i \right) \\ & + \sum_{i \in \overline{\mathcal{G}_{\text{B}}}} \sum_{j \in (\mathcal{P}_i \cap \mathcal{G}_{\text{B}})} K_S \cdot X_{j,i} \cdot x_j \end{aligned} \tag{7.33}$$

The constant G_i describes the execution time difference when basic block i is executed from the private memory when compared to the shared memory. This

constant is estimated for each basic block by initially placing all basic blocks into the private memory and performing a timing analysis. X_i is the worst-case execution count of basic block i taken from the last timing analysis done. The worst-case execution count refers to number of times a basic block is executed on the path leading to the overall worst-case execution time. Similarly, $X_{j,i}$ describes how many times the transition from basic block j to i was executed on the path leading to the WCET. The variable $k_{j,i}$ describes the estimated number of additional cycles due to jump correction code for the jump from basic block j to i. It is defined as follows:

$$k_{j,i} = K_{\mathrm{S}} \cdot (\overline{x}_j \wedge x_i) + K_{\mathrm{P}} \cdot (x_j \wedge \overline{x}_i) \tag{7.34}$$

Where K_{S} is the estimated number of cycles required for additional jump correction code when executing in the shared memory and K_{P} is the corresponding value when executed from private memory. The term $x_i \cdot G_i \cdot X_i$ in Equation (7.33) describes the (optimistic) estimated decrease in WCRT by moving the basic block i into the private memory. The subtractive sum term $k_{j,i} \cdot X_{j,i}$ represents the estimated additional timing increase due to jump correction code to be executed when block i and its predecessor j are not placed in the same memory region. The term $K_P \cdot X_{j,i} \cdot x_i$ in Equation (7.33) denotes the gain due to the removal of obsolete jump correction code by placing basic block i in the private memory while its predecessor j was already assigned there during a previous iteration. Similarly, the last additive term of Equation (7.33) estimates the gain in timing due to the removal of unnecessary jump correction code in case basic block i was assigned to the private memory in a previous iteration and its predecessor j would be moved there as well.

Equation (7.33) implicitly considers the user-defined constant N_{M}. As the ILP neither models the control-flow graph, nor the actual worst-case execution path of the program, the modeling of the estimated timing gain relies on the last timing analysis carried out and the derived worst-case execution count X_i per basic block i. The larger N_{M} is chosen, the more basic blocks are allocated in a single iteration without evaluating its effects in between. As the WCEP could switch with moving a single basic block, a larger value of N_{M} may lead to unnecessary basic blocks being assigned to the scarce private memory, as a part of the newly assigned basic blocks may not lead to a reduction in WCRT.

Beside estimating the core's own gain, the ILP used in the *getBestBB()* function also models a *cooperative* gain, which represents the estimated timing gain of tasks on concurrent cores by the allocation decisions of the current task. Similar to the idea of the cooperative gain presented in the previous optimization approach, the underlying idea is that when allocating a potentially frequently executed basic block to the private memory, the decreased bus contention can also improve the timing guarantees for tasks on other cores. This cooperative gain is denoted as the ILP variable ν here. It is determined by accumulating the estimated timing gain for the concurrent cores' tasks of moving a basic block to the private memory. The constant $U_i^{n,m}$ describes the estimated timing gain for task τ_m by moving the basic block i of task τ_n to the private memory. The cooperative gain ν of a task τ_n is determined by the following equations:

$$\nu = \sum_{\tau_m \in \Gamma, \tau_m \neq \tau_n} \sum_{i \in \mathcal{G}_{\mathrm{B}}} U_i^{n,m} \cdot x_i \tag{7.35}$$

$$U_i^{n,m} = \begin{cases} 0 & \text{if } (Y_n = 0) \text{ or } (Y_m = 1), \\ F_S \cdot Z_i^{n,m} & \text{else if } (Z_i^{n,m} > 0), \\ \theta_i & \text{else.} \end{cases} \tag{7.36}$$

$$Y_n = \begin{cases} 1 & \text{if } D_n \geq R_n^+, \\ 0 & \text{else.} \end{cases} \tag{7.37}$$

$$Z_i^{n,m} = \begin{cases} \theta_i & \text{if } A_{E,m}^+ \geq \eta_n^+(R_m^+), \\ A_{E,m}^+ - \eta_n^+(R_m^+) + \theta_i & \text{else.} \end{cases} \tag{7.38}$$

The positive timing gain for the other cores by moving a basic block i of task τ_n to the private memory is accumulated by Equation (7.35). Note that the Equations (7.36) to (7.38) only contain constants and are calculated beforehand, hence they are not existing as constraints inside the ILP model. $U_i^{n,m}$ is set to 0 in case the current task τ_n (for which the ILP is modeled) does not meet its deadline yet ($Y_n = 0$) or the concurrent task τ_m does already meet its deadline ($Y_m = 1$, cf. Equation (7.37)). This reflects the idea that the optimization tries to reach a state in which the current task τ_n does not violate its deadline first (as the cooperative gain ν will always be 0 in this case). Similarly, if task τ_m already meets its deadline, task τ_n does not need to "help" this task, as it already met its aim.

If the current task τ_n does already meet its deadline and task τ_m does not, $U_i^{n,m}$ is potentially non-zero. The idea of $U_i^{n,m}$ is to estimate the number of effectively reduced competing bus accesses for task τ_m when moving basic block $\mathtt{B}_i$ of task τ_n to the private memory. This value is represented by $Z_i^{n,m}$ and defined in Equation (7.38). Each effectively reduced competing bus access can optimistically decrease the other task τ_m's WCRT by F_S cycles (the shared memory's net access latency, cf. Equation (7.17) (Page 169)), hence $Z_i^{n,m}$ is multiplied by F_S. $A_{E,m}^+$ denotes the maximum number of bus accesses performed by task τ_m in one complete task execution, whereas $\eta_n^+(R_m^+)$ represents the maximum number of bus accesses by task τ_n during the execution time of task τ_m. θ_i denotes the number of access contributed by the object (i.e., a basic block or data object) i of task τ_n along the sub-path leading to the maximum number of bus accesses $\eta_n^+(R_m^+)$ performed by task τ_n during the execution of task τ_m. Simply put, the first case of Equation (7.38) represents that if task τ_m performs at most more or the same number of bus accesses in a complete task execution than task τ_n does at most during the execution of task τ_m, the number of effectively reduced bus accesses when moving the object i to the private memory equals how many accesses are contributed by this object. In case the concurrent task τ_m's maximum number of bus accesses during one complete task execution is less than the maximum number of bus accesses performed by task τ_n during the execution of task τ_m, $Z_i^{n,m}$ is set to the term $A_{E,m}^+ - \eta_n^+(R_m^+) + \theta_i$. The difference $A_{E,m}^+ - \eta_n^+(R_m^+)$ describes how many bus accesses task τ_n may perform *more* during the execution of task τ_m compared to the number of bus accesses of task m. Since $A_{E,m}^+$ is lower than $\eta_n^+(R_m^+)$ for this case to be selected, this difference always has a negative sign. Thus, θ_i needs to be larger than $|A_{E,m}^+ - \eta_n^+(R_m^+)|$ for the overall term to become positive. In this case, the number of reduced concurrent bus accesses θ_i by moving object i is large enough to also *effectively* reduce the number of competing bus accesses for task τ_m. This is due to the circumstance that if a task τ_m has at most $A_{E,m}^+$ bus accesses during a complete

task execution, no more bus accesses than these can be blocked. In the following, two examples are given two highlight the reasoning for these two cases of Equation (7.38).

Example 7.3. *A multi-core system as depicted in Figure 2.6 (cf. Page 13) with two cores and one task allocated to each core is given. Task τ_0 is allocated to first core, whereas τ_1 is assigned to the second core. The bus is arbitrated following a round robin strategy. Task τ_0 performs at most 100 bus accesses during a complete task execution ($A^+_{E,0}$). Task τ_1 accesses the bus at most 50 times during the concurrent execution of task τ_0 ($\eta^+_1(R^+_0)$). Therefore, $A^+_{E,0} \geq \eta^+_1(R^+_0)$ holds. Up to 50 accesses of task τ_0 are competing with accesses from task τ_1. The analysis of $\eta^+_1(R^+_0)$ also details that 10 of these 50 bus accesses performed at most during the execution of task τ_0 are due to the instruction fetches of a basic block B_i of task τ_1 (θ_{Bi}). By moving basic block B_i to the private memory, the number of competing bus accesses for task τ_0 could be reduced by up to these 10, leading to only 40 accesses of task τ_0 to be potentially competing with task τ_1.*

Example 7.4. *The same basic multi-core architecture as in the previous example is assumed with one task allocated to each core. Task τ_0 again performs at most 100 bus accesses during a complete task execution ($A^+_{E,0}$). Yet, τ_1 accesses the bus at most 120 times during the concurrent execution of task τ_0 ($\eta^+_1(R^+_0)$) this time. Therefore, $A^+_{E,0} \geq \eta^+_1(R^+_0)$ does not hold. Again, it is assumed that 10 of the maximum bus accesses of task τ_1 are due to instruction fetches of basic block B_i. In this case, moving basic block B_i of task τ_1 to the private memory does not reduce a single potentially competing access for task τ_0. This is due to the fact that, since task τ_0 performs at most 100 accesses during a complete task execution, at most 100 accesses of τ_0 can be blocked due to competing accesses. As the moving of basic block B_i of task τ_1 to the private memory only reduces the number of bus accesses performed by task τ_1 during the execution of task τ_0 to $\eta^+_1(R^+_0) = 110$ at best, τ_0 still has at most 100 blocked accesses. Hence, $\eta^+_1(R^+_0)$ needs to be reduced by at least 21 accesses to* effectively *reduce the number of competing accesses for task τ_0. This is expressed using the term $A^+_{E,m} - \eta^+_n(R^+_m) + \theta_i$ in the last case of Equation* (7.38).

$Z^{n,m}_i$ becomes negative or zero in case that assigning the basic block B_i of task τ_n does not reduce the effective number of competing accesses of task τ_m (see the previous example). In this case, $U^{n,m}_i$ becomes a figure of merit, indicating that by moving basic block B_i to the private memory, θ_{Bi} accesses may be reduced, but by moving B_i alone, the number of competing accesses for task τ_m cannot be reduced. The reason behind this is to guide the heuristic to allocate the most promising basic blocks, even if in the current iteration, the competing accesses for other tasks may not be effectively reduced. Additionally, the number of effectively competing accesses for a task may still be reduced after all, as $Z^{n,m}_i$ is determined on a basic block level and does not include the effects of other basic blocks being allocated as well in the current iteration.

In case the current task τ_n already meets its deadline in the current iteration, an additional constraint is inserted to ensure that the task's WCRT stays below its deadline:

$$\mu + D_n - R^+_n \geq 0 \tag{7.39}$$

Since the task does not violate its deadline in the current iteration, $D_n - R^+_n$ has to be greater than or equal to 0. Yet, the task's own gain μ may be negative, when

basic blocks are assigned to the private memory to improve the cooperative gain, leading to new jump correction code and potentially worsen the task's WCRT. Such a degradation in the timing for the sake of a higher cooperative gain is permitted, as long as the task's own deadline does not become violated, which is described by Equation (7.39).

Eventually, the ILP's objective function is inserted. In case the task does not meet its deadline yet ($Y_n = 0$), the ILP is set to maximize the task's own gain μ (as the cooperative gain is anyway set to 0 in this case). Otherwise, the ILP maximizes the estimated cooperative gain ν:

$$\max : (1 - Y_n) \cdot \mu + \nu \tag{7.40}$$

getBestData() This function used in the description of the greedy heuristic shown in Algorithm 7.1 should return the most promising N_M data objects of task τ_n to improve the system's timing. In contrast to the previously discussed function, the functionality of *getBestData()* is not implemented via an ILP, but rather by a straightforward greedy first fit algorithm. This function is not implemented using an ILP, as the allocation decisions for data objects have less complex side effects, as, e.g., no potential additional code needs to be added.

The overall algorithm is described in Algorithm 7.2. First, the set $\mathcal{G}_\mathrm{D}$ is created in Lines 2 to 7 which holds all data objects which are currently allocated to a shared memory. The part of the algorithm spanning from Line 8 to 24 initializes a sorted map **A** which is sorted by its keys' values (from high to low). The map uses data objects as keys and the potential gain by moving these data objects to the private memory as the corresponding values. In case the current task does not meet its deadline yet ($Y_n = 0$), the algorithm prioritizes to improve the task's very own timing. The two nested loops beginning in Lines 9 and 10 iterate over all basic blocks of the current task τ_n and all data objects accessed in each of these basic blocks. Here, the set $\mathcal{D}_n^i$ contains all data objects accessed during the execution of basic block i which is part of task τ_n. Only data objects which are currently not yet assigned to the private memory are considered. The potential timing gain for the task itself is determined and added in Line 12. $\mathbf{A}[k]$ contains the estimated timing gain for moving the data object k into the private memory. The term $X_i \cdot G_k^i$ describes the net timing gain G_k^i of moving the data object k to the private memory during the execution of basic block i, multiplied by the worst-case execution count X_i of basic block i during the last timing analysis. The net timing gain G_k^i is derived as shown in Equation (7.5) (cf. Page 167) and describes the timing gain due to the net memory access latency difference between the shared and private memory. T_k^i describes the accumulated bus stall times of the accesses to data object k during the execution of basic block i as determined by the last timing analysis.

In case the current task τ_n already meets its deadline ($Y_n = 1$), the cooperative gain for moving data object k into the private memory is estimated. This is done by iterating over all data objects of the current task in Lines 17 to 23 and accumulating the estimated timing gain for each concurrent task τ_m (cf. Line 18) by moving the data object k in the private memory. Similar as before, only data objects which are not already assigned to the private memory are considered here (cf. Line 19). Here, $U_k^{n,m}$ denotes the estimated timing gain for task τ_m when data object k from task τ_n is

Algorithm 7.2 getBestData()

Inputs: τ_n – Task to be optimized; Γ – Set with all tasks; N_M – Maximum number of basic blocks or data objects allocated to each SPM per iteration
Output: $\mathcal{R}$ – A set containing data objects to be allocated

1: $\mathbf{A} \leftarrow \emptyset$ *// Map* $\mathbf{A}$ *is sorted by its keys' values (from high to low).*
2: $\mathcal{G}_D \leftarrow \emptyset$
3: **for** $k \in \mathcal{D}_n$ **do**
4: **if** k is in shared memory **then**
5: $\mathcal{G}_D \leftarrow \mathcal{G}_D \cup \{k\}$
6: **end if**
7: **end for**
8: **if** $Y_n = 0$ **then**
9: **for** $i \in \mathcal{B}_n$ **do**
10: **for** $k \in \mathcal{D}_n^i$ **do**
11: **if** $\{k\} \subseteq \mathcal{G}_D$ **then**
12: $\mathbf{A}[k] \mathrel{+}= X_i \cdot G_k^i + T_k^i$
13: **end if**
14: **end for**
15: **end for**
16: **else**
17: **for** $k \in \mathcal{D}_n$ **do**
18: **for** $\tau_m \in \Gamma \backslash \{\tau_n\}$ **do**
19: **if** $\{k\} \subseteq \mathcal{G}_D$ **then**
20: $\mathbf{A}[k] \mathrel{+}= U_k^{n,m}$
21: **end if**
22: **end for**
23: **end for**
24: **end if**
25: $t \leftarrow 0$
26: $\mathcal{R} \leftarrow \emptyset$
27: **for** $k \in \mathbf{A}$ **do**
28: **if** $S_{DSPM} - W_{DSPM} - t > S_k$ **then**
29: $\mathcal{R} \leftarrow \mathcal{R} \cup \{k\}$
30: $t \leftarrow t + S_k$
31: **end if**
32: **if** $(|\mathcal{R}| = N_M)$ or $(S_{DSPM} - W_{DSPM} - t = 0)$ **then**
33: break
34: **end if**
35: **end for**
36: **return** $\mathcal{R}$

moved to the private memory. This value is determined using the same approach as shown in Equation (7.35) for basic blocks.

After the first part of the algorithm, map $\mathbf{A}$ is filled with the data objects and the corresponding estimated timing gains for moving them in the private memory. The temporary variable t in Line 25 is a counting variable, holding the number of

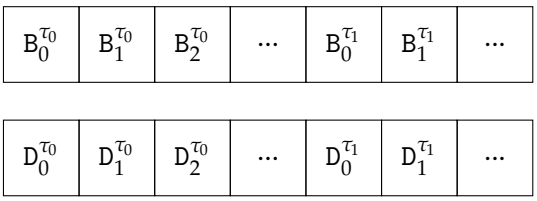


Figure 7.3. – The genome of a single individual represented by two lists, containing the allocation decisions of each object, grouped by tasks.

bytes allocated by data objects assigned to the private memory in this iteration. The temporary set $\mathcal{R}$ is used to hold all data objects that are assigned during the current iteration. The loop spanning over Lines 27 to 35 iterates over the keys of map $\mathbf{A}$ in a descending order of the associated timing gain previously determined per data object (key). It is checked whether the space left over in the data SPM is large enough to accommodate the current data object in Line 28, where W_{DSPM} represents the number of bytes already allocated in the data SPM in previous iterations. If the data object still fits, it is added to the set $\mathcal{R}$ and its size is added to the counting variable t (cf. Lines 29 and 30). In case that N_{M} data objects have been assigned during this iteration or the data SPM has become completely full, the loop is exited early. Finally, the algorithm returns the set $\mathcal{R}$ which contains the data objects to be allocated.

7.4 Evolutionary Algorithm

Evolutionary algorithms are a common approach for solving complex problems for which a precise mathematical description is not existing. In contrast to the previous purely ILP-based approach presented in Section 7.2, the evolutionary algorithm has the potential advantage to precisely measure the effect of each change in the allocation, and it does not rely on a somewhat simplified model, as each possible solution is evaluated by a complete timing analysis. In contrast to the greedy heuristic presented in the previous section, the evolutionary algorithm can also undo allocation decisions due to recombination or mutation operations. The general concept of the evolutionary algorithm is based on the evolutionary algorithm presented for performing a bus-aware static instruction SPM allocation for TDMA-arbitrated systems [OLF17]. It is extended to support also the allocation of data objects. Besides, the fitness calculation and the mutation process were modified as well. The initial population is created in the same manner as described in the related work [OLF17]. Here, one individual of the initial population is created with all basic blocks and data objects assigned to the shared memories, resembling the default case. For the remaining individuals of the initial population, all basic blocks and data objects are assigned to the private memory. Since this will lead to overflowing private memories, a repair function is called for each individual which randomly allocates program parts back to the shared memory. In the following, the genome composition of a single individual, the fitness function, the crossover of two individuals and the repair process are described.

7.4.1 Genome Composition

A single individual is represented using two lists of Boolean values and contains the allocation decisions for all basic blocks and data objects for all tasks of a system. The first list contains the allocation decisions for all basic blocks of each task in the system and has therefore the length of $|\mathcal{B}|$. More precisely, the allocation decisions for all basic blocks of a task are grouped together, leading to a concatenation of the basic block allocation decisions for each task. The second list contains one Boolean value per data object for each task, grouped per task as well, representing whether a data object should be allocated to the SPM or not. This principle is illustrated in Figure 7.3.

7.4.2 Fitness Calculation

The fitness of a single individual is derived by accumulating the quotient of the response time and corresponding deadline of the task of each core. With R_n^+ being the WCRT of a task τ_n and D_n being the corresponding deadline, the fitness of an individual i, denoted as ζ_i, is determined using the following equation:

$$\zeta_i = \sum_{\tau_n \in \Gamma} \begin{cases} \frac{R_n^+}{D_n} & \text{if } R_n^+ > D_n, \\ 0 & \text{else.} \end{cases} \tag{7.41}$$

If an individual i's fitness ζ_i is lower than an individual j's fitness ζ_j, the individual i is considered fitter than individual j. As soon as an individual with a fitness of 0 is found, the algorithm is terminated, as this corresponds to a solution where all tasks are meeting their deadlines. All fitness values are additionally cached. Therefore, the fitness value of an individual does not need to be re-calculated (which requires a timing analysis of the whole system) in case its allocation decisions are equal to a previous individual, reducing the overall required algorithm runtime.

7.4.3 Mutation

At the very beginning of each new generation the mutation phase is executed. Here, every allocation decision is iterated and potentially flipped with a user-defined probability of P_M.

7.4.4 Repair Function

After the mutation phase of each new generation, a repair function is called for each individual. As the genome is simply represented by two lists with Boolean values representing the allocation decisions for each basic block or data object, many potential solutions are not valid, as the number of allocated objects to the private memory exceeds the corresponding memory capacity. Naively discarding these solutions would slow down the algorithm's convergence, as, e.g., good solutions which only exceed a private memory by a few bytes would be completely ignored, although they could be easily repaired. Therefore, the repair function checks at first, if the current solution exceeds the given memory limits. If so, objects allocated to the overflown memory are randomly chosen and re-assigned to the shared memory until the individual's allocation decisions are valid again.

Subsequently, the repair function also performs a jump correction on each individual, inserting jump correction code where otherwise the control-flow would be broken. As this may insert additional code into the private instruction memories, the repair function starts over and verifies that no memories are overflown. This iterative process is repeated until the control-flow is valid and none of the private memories are overflown.

7.4.5 Crossover

The crossover of two individuals is performed by a single point crossover. Therefore, two random numbers are drawn (uniformly distributed), the first in the range of $[0, |\mathcal{B}| - 1]$ and the second in the range of $[0, |\mathcal{D}| - 1]$. The first point represents the crossover point for the list of basic blocks, where the new individual takes over the basic block allocation decision of the first parent individual up to the found crossover point and the remaining allocation decisions are adopted from the second parent individual. The second number drawn represents the crossover point for the data object allocation decisions. Here, the crossover of the data object lists is done analogously as described for the basic blocks.

7.5 Evaluation

The following section evaluates the presented approaches and discusses the results. The evaluation setup is described in the upcoming section.

7.5.1 Evaluation Setup

The initially presented architecture shown in Figure 2.6 (cf. Page 13) is assumed as the basic architecture for the evaluation, where each core has private instruction and data SPMs, as well as shared instruction and data memories. Systems with 2, 4 and 8 cores are evaluated. The SPM of each core is resized to a certain percentage of the benchmark allocated to the core in order to see the effect of the optimization in dependence of the available memory to allocate. SPM sizes of 20%, 40%, 60% and 80% were evaluated, where an SPM size of 20% refers to an instruction SPM large enough to hold 20% of a task's code and a data SPM with a size of 20% of a task's data objects. A round robin-based bus arbitration is assumed. As mentioned before, the presented optimizations in this chapter are not limited to a round robin-based bus arbitration, but could also be applied to other work-conserving arbitration schemes (also to non work-conserving arbitration, e.g., TDMA, yet the cooperative character would not have any effect there) with adjustments. For the purely ILP-based approach, the accumulated worst-case bus blocking time for a task L_i, as well as the worst-case delay of a single bus access π would have to be derived for the corresponding bus arbitration. For the greedy heuristic, the number of effectively reduced bus accesses $Z_i^{n,m}$ would have to be adapted for the corresponding bus arbitration. The evaluation for other bus arbitration schemes is reserved for future work.

The net latency of the shared memories is assumed to be 6 cycles (similar to the default setting of existing embedded systems [NXP09a, Inf07]), whereas an SPM access is assumed with a single cycle latency. Each core has a single task allocated,

activated with a periodic activation pattern. The period of each task is determined according to an initial core utilization: First, a pessimistic WCRT of each task is derived by assuming the worst-case latency for each shared memory access with all instructions and data objects being allocated to the shared memory. The reason for assuming a worst-case latency for each bus access is that for a tighter timing analysis, all task periods need be known (as the number of potentially competing accesses for a task depends on the other tasks' periods). Yet, to have a comparability between the different systems, a task's period P should be set according to its unoptimized WCRT R^+ and the desired utilization U via $P = R^+/U$. This would inflict a cyclic dependency between the WCRT and the period of a task, hence the worst-case latency assumption for deriving the unoptimized WCRT and the task's period. The period of each task is therefore set to the WCRT divided by the desired initial core utilization. This simply sets a common baseline for all systems and does not change anything for the optimizations, timing analyses used within the optimizations or to validate the final result. All tasks are assumed to have implicit deadlines, meaning their deadline is equal to their period. Initial core utilizations from 1.0 to 3.0 in steps of 0.2 were evaluated. Neither the task activation pattern, nor the deadline assumption are required restrictions of the presented optimizations and can be replaced by any other. Benchmarks of the following benchmark suites were evaluated: MRTC [GBEL10], MediaBench [LPM97], StreamIt [Str18], and UTDSP [LCS92]. Additionally, a set of miscellaneous benchmarks, mostly consisting of de- and encoders were evaluated as well. Benchmarks from this set were randomly ordered and grouped by using a sliding window over the list. This results in a total number of 86 different systems to be evaluated for each multi-core setting. See Appendix E.2 for a detailed overview of all evaluated benchmark configurations. Overall, 4 different approaches are evaluated: The ILP-based cooperative combined allocation, the greedy heuristic, the evolutionary algorithm-based allocation, as well as the completely bus-unaware ILP-based allocation based on the related work ([SMRC05], [FK09]) as a reference (cf. Section 7.2.1, Page 165). Since the reference bus-unaware ILP-based allocation is always done as a pre-requisite of the bus-aware ILP-based cooperative combined allocation (cf. Section 7.2), the reference approach does not need to be evaluated separately. With the different number of cores per system, SPM sizes, initial utilizations per core, benchmark sets and optimization approaches combined, the total number of settings evaluated exceeds 34 000 (dual-, quad- and octa-core architectures, 86 benchmark configurations per architecture, 3 different optimizations, 11 different utilizations, 4 different SPM settings).

All experiments were performed on an Intel Xeon Server (48 cores at 3.2 GHz with 1.48 TB RAM) and ILPs were solved using Gurobi 8.1.0, whereas each ILP solving process was limited to 1 thread. All benchmarks were compiled with several ACET-oriented optimizations activated (`-O2` optimization flag of the WCC compiler, cf. Chapter 3). A general timeout of 1 h per core is used for the optimization. Timing analyses were done using the principles of a semi-integrated timing analysis discussed in Section 6.5.3.

7.5.2 Dual-Core Evaluation

Figure 7.4 shows the results for dual-core systems with varying SPM sizes. Each graph depicts the results for a specific SPM size setting, ranging from 20% at the very

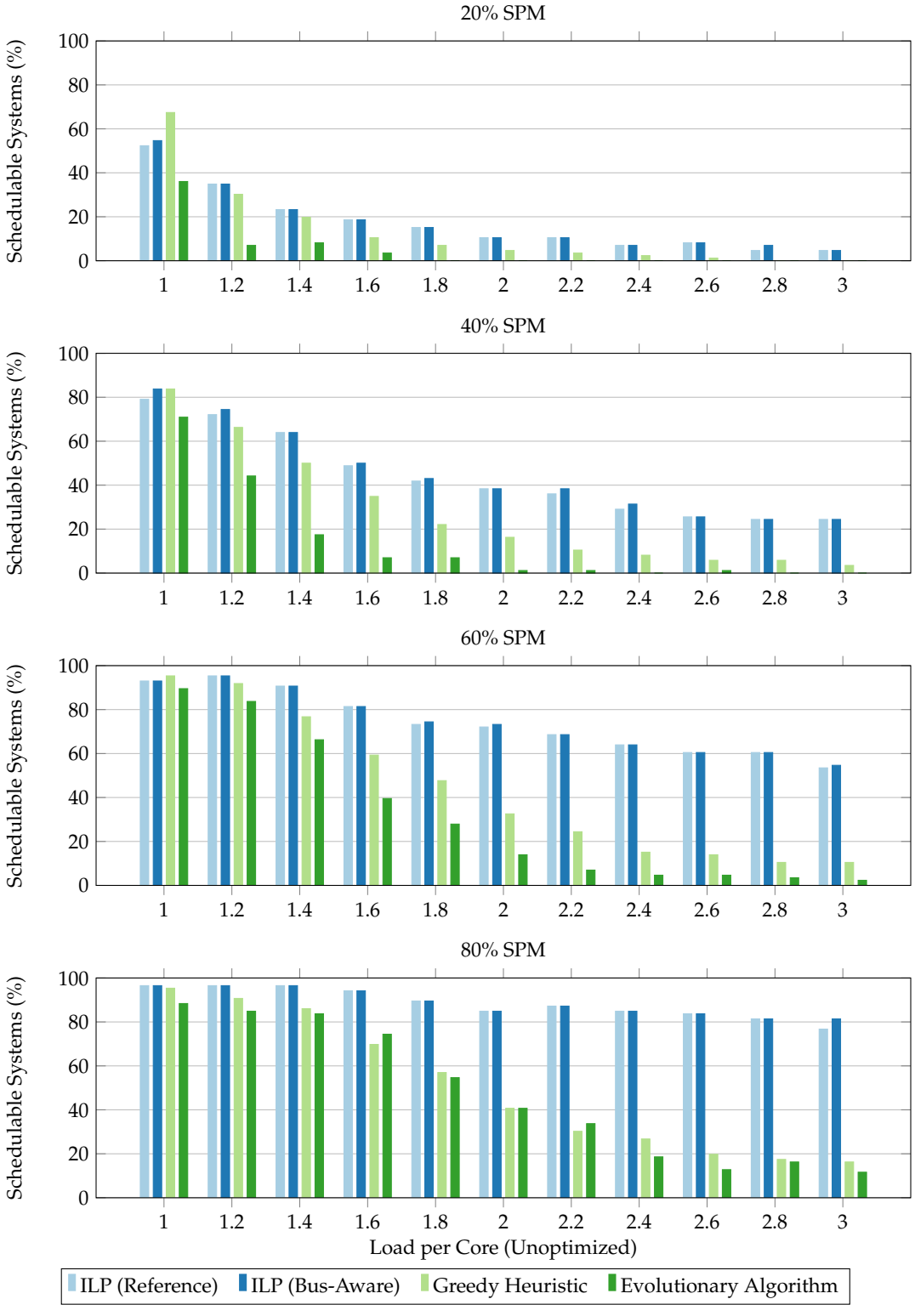


Figure 7.4. – Percentage of schedulable systems after applying the memory allocation optimization dependent on the initial load per core and SPM size for dual-core systems.

top to 80% at the very bottom. The x-axis of each graph shows the initial utilization per core, ranging from 1.0 to 3.0, whereas the percentage of schedulable systems for each configuration is marked on the y-axis. A y-axis value of 40% at an x-axis value of 2.0 denotes that of all systems evaluated for this specific setting, 40% of these are schedulable after applying the optimization, while the initial task WCRTs were twice their respective deadlines. The different bars per group represent the different optimization approaches. *ILP (Reference)* refers to the bus-unaware reference allocation from Section 7.2.1.

As expected, the overall number of schedulable systems decreases with an increasing initial utilization, independent of the optimization. For a 20% SPM setting, the greedy heuristic generates the largest number of schedulable systems for the lowest utilization rate. In case of greater utilization rates, the greedy heuristic falls below the performance of the fully ILP-based methods, yet always above the evolutionary-based approach. The evolutionary algorithm consistently delivers the lowest number of schedulable systems and is not able to create any schedulable system for utilization rates of 1.8 or higher. The bus-aware cooperative ILP-based method is not able to improve over the reference ILP approach for this configuration in most cases, with minor exceptions at utilization rates 1.0 and 2.8.

Increasing the SPM to 40% results in overall more systems becoming schedulable also for greater utilization rates. The overall picture is very similar to the smaller SPM setting, yet with a few exceptions. The greedy heuristic is able to create the same number of schedulable systems as the bus-aware ILP approach for the lowest utilization rate, yet drops below for higher utilization rates. Additionally, the cases where the bus-aware cooperative ILP approach yields improvements over the reference ILP increased. Though the additional number of systems becoming schedulable due to the bus-awareness and cooperative character is minor, this improvement can be seen for many utilization rates.

When comparing the evaluation results of the 60% SPM setting to the previous 40%, only a few differences beside the overall greater number of schedulable systems can be noticed. The greedy heuristic is able to generate the largest number of schedulable systems for the lowest utilization rate, yet for larger utilization rates drops below the fully ILP-based approaches as seen before. The number of cases where the bus-aware cooperative ILP approach can improve over the reference ILP is slightly decreased compared to the 40% SPM setting. Yet still, there are several cases where the presented bus-aware cooperative ILP approach can achieve a few more schedulable systems than the reference approach.

The trend of a decreasing improvement of the bus-aware cooperative ILP approach over the reference ILP with an increasing SPM size continues for the 80% SPM setting. At a 60% SPM setting, the newly presented ILP-based approach can slightly improve compared to the reference for utilization rates 1.8, 2.0 and 3.0, while at an 80% SPM setting the bus-aware ILP only shows an improvement at a utilization of 3.0. The greedy heuristic is not able to gain a clear improvement over any of the other approaches for this large SPM setting anymore at any utilization rate. An exception to the previous SPM settings can be seen for the evolutionary algorithm. For utilization rates 1.6 and 2.2, the EA generates a larger number of schedulable systems than the greedy heuristic. A possible reason for the evolutionary algorithm performing comparably better for a larger SPM setting here is that the likelihood for the initial

individuals of being closer to the optimal solution is greater in case of a larger SPM. All initial individuals except one have all basic blocks placed into the SPM and are then repaired by randomly moving excess blocks to the shared memory until the SPM does not overflow anymore. Since the number of blocks to be removed from the initial allocation decreases with a growing SPM, the chance of an initial individual being close to an optimal solution increases.

Overall, the evaluation of the dual-core systems showed that even for multi-core systems with a small number of cores, the bus-aware cooperative approaches can lead to improved allocations.

7.5.3 Quad-Core Evaluation

The results for the evaluated quad-core systems are shown in Figure 7.5. The structure of the graphs is identical to the ones discussed for the dual-core systems. As expected, the overall trend of a decreasing number of schedulable systems with an increasing load per core is also visible for the quad-core systems. Compared to the evaluation results of the dual-core systems, the bus-aware ILP model and also the greedy heuristic outperform the bus-unaware ILP-based reference allocation for a larger number of configurations. For the smallest SPM configuration evaluated at 20%, the greedy heuristic outperforms the next best approaches for initial utilizations from 1.0 to 1.4. At an initial per core utilization of 1.0, the greedy heuristic is able to repair 73.26% of the systems, while the reference ILP approach only reaches 46.51%. The bus-aware ILP-based optimization is able to improve the number of repairable systems slightly compared to the bus-unaware model at a few utilization settings, but only in the range of 1% to 4%. The evolutionary-based approach consistently performs the worst and is not able to repair any system for utilization rates higher than 1.0.

With an increasing SPM size of 40%, the effectiveness of the greedy heuristic compared to the ILP-based approaches degrades. While the approach is close to the results of the bus-aware ILP model and is better than the reference ILP for utilization rates 1.0 and 1.2, its performance quickly deteriorates for utilization rates of 1.4 and greater. At a 40% SPM setting, the bus-aware ILP-based approach achieves a greater number of schedulable systems for 7 out of the 11 evaluated utilization rates. It results in up to 8.4% more systems in total becoming schedulable after applying the optimization.

As already seen when increasing the SPM size from 20% to 40%, overall more systems become schedulable when increasing the SPM further to 60% as expected. The greedy heuristic slightly outperforms the ILP-based approaches for the lowest utilization rate evaluated. For utilization rates of 1.4 and higher, the greedy heuristic drops in performance when compared to the ILP-based approaches. Compared to the 40% configuration, the bus-aware ILP model outperforms the bus-unaware reference ILP model in a smaller number of occasions for the 60% SPM configuration. For most utilization rates, the same amount of systems become schedulable after applying an ILP-based optimization, irrespective of whether the ILP model is bus-aware or not. Yet for utilization rates 1.8, 2.0, 2.6 and 2.8, the bus-aware ILP model is able to render more systems schedulable than any other approach.

The evaluation results for the quad-core systems with an 80% SPM setting follow the same trends as seen for the previous increasing SPM sizes. The greedy heuristic is not able to improve on the fully ILP-based approaches for any utilization rate

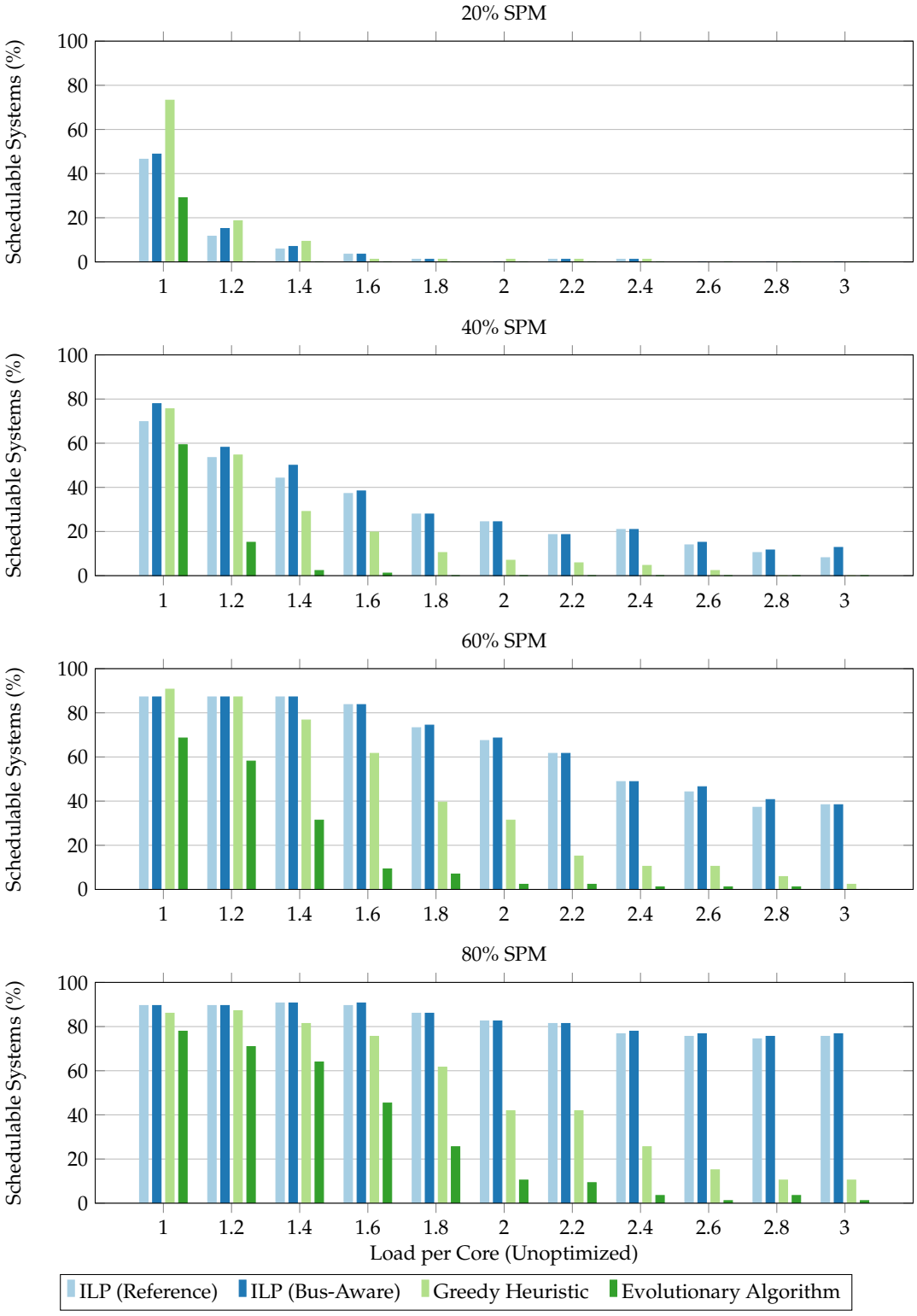


Figure 7.5. – Percentage of schedulable systems after applying the memory allocation optimization dependent on the initial load per core and SPM size for quad-core systems.

anymore. Similar to the previous results, the bus-aware ILP-based approach does not result in a higher number of schedulable systems for utilization rates lower than 2.4 (with an exception at 1.8). For all evaluated utilization rates equal to this or higher, the bus-aware ILP-based approach is able to slightly outperform the bus-unaware ILP-based approach.

Overall, some insights can be gathered for the quad-core systems. For very small SPMs or comparably low utilization rates, the greedy heuristic tends to perform the best out of all approaches. A possible reason for this is the lower pessimism of the greedy heuristic, as all decisions made in one iteration are precisely evaluated before the next iteration. This may lead to a higher usage of the very small SPM, as potentially over-estimated additional size costs due to jump corrections are less likely, as each iteration step only needs to consider the effects of a smaller number of decisions. For SPMs settings which can hold around the half of a task, the bus-aware ILP-based optimization tends have the best performance. In case the SPM can hold nearly an entire task, the additional bus-awareness of the ILP-based optimization pays off only for higher utilization rates. A possible reason for this is that if the entire task nearly fits into the SPM, the bus-unaware ILP-based approach will place the majority of the task into the SPM in most cases. This reduces the difference between the bus-aware ILP-based allocation and the bus-unaware one.

7.5.4 Octa-Core Evaluation

The evaluation results for the octa-core systems are shown in Figure 7.6. The structure of the graphs is identical to the ones discussed previously. Overall, the trend of a smaller number of schedulable systems after applying an allocation optimization with an increasing number of cores can be seen here as well. This is caused by two different reasons: First, with an increasing number of tasks inside a system, the chance of having a task with a very low optimization potential in this system increases, which increases the chance of the system not being schedulable after the optimization. Second, while the timeout limit is linearly increased with the number of cores, the number of analysis steps inside the timing analysis grows with $\mathcal{O}(N_C^2)$, where N_C is the number of cores inside a system (as for each task, the number of competing bus accesses have to be derived). This also increases the chance of an evaluation being canceled due to a timeout.

For the 20% SPM setting, none of the approaches are able to create a schedulable system for utilization rates greater than 1.2. The greedy heuristic outperforms the other approaches for a utilization rate of 1.0 and is on par for 1.2 Also, the bus-aware ILP-based method can outperform the ILP-based reference approach for a utilization rate of 1.0. As seen in the previous evaluation results, the evolutionary-based approach shows the worst performance for all configurations. While it is able to generate a few schedulable systems for a utilization rate of 1.0, it cannot repair any system for 1.2.

When increasing the SPM size to 40%, the overall effectiveness of the greedy heuristic seems to degrade as already noticed for the dual-core and quad-core systems. The greedy heuristic is able to outperform the other approaches for a utilization rate of 1.2 and is on par with the reference ILP approach for 1.4, yet performs worse than the fully ILP-based approaches for higher rates. The bus-aware ILP approach yields significantly better results for utilization rates 1.0 and 1.2, and slightly better ones for

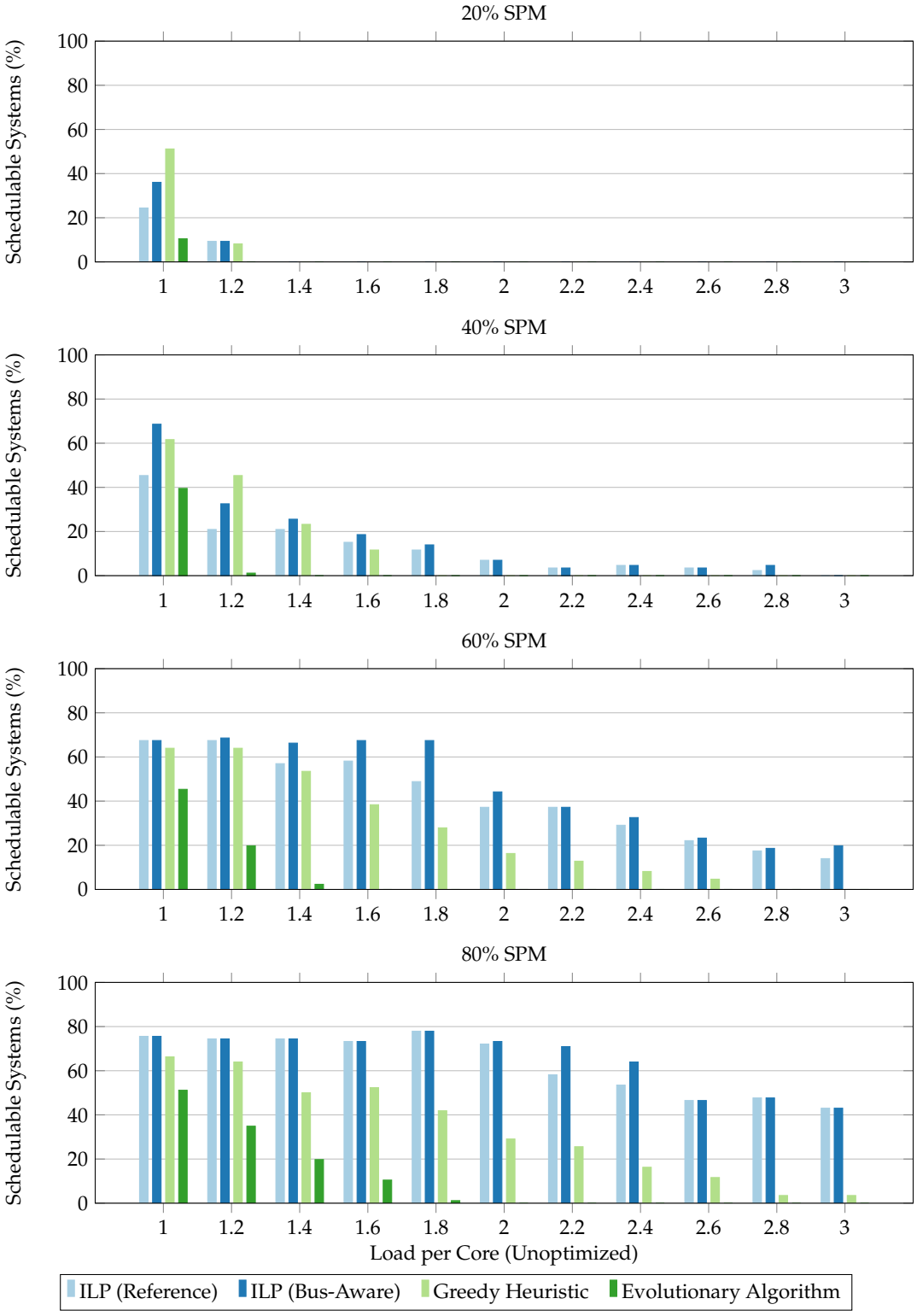


Figure 7.6. – Percentage of schedulable systems after applying the memory allocation optimization dependent on the initial load per core and SPM size for octa-core systems.

many of the higher rates when compared to the reference ILP model. Compared to the results of the quad-core systems evaluated with the same SPM setting, the margin by which the bus-aware ILP method outperforms the bus-unaware one is increased. Whereas the bus-aware ILP method resulted in up to 11.63% more schedulable systems than the reference model for the quad-core systems, this increased to 23.26% for the octa-core systems.

For the 60% SPM setting, the bus-aware ILP approach shows a significant improvement for several utilization rates. At a utilization rate of 1.8, the bus-aware ILP approach can repair 67.44% of all systems, whereas the reference ILP only results in 48.84%. The greedy heuristic is on par with the reference ILP approach for the lowest 3 of the evaluated utilization rates, while it consistently performs worse for higher rates. Furthermore, anomalies can be seen for both ILP-based approaches at utilization rates 1.4 and 1.6. Here, *more* systems become schedulable than compared to the next *lower* utilization rate. All of the additional schedulable systems for the larger utilization rates are due to ILP-related terminations of the evaluations with lower utilization rates during a timing analysis. Not only does the number of ILPs to solve grow during the timing analysis for all tasks with an increasing number of cores, but also the WCRTs increase due to a higher bus contention. These larger coefficients and variable value ranges inside the ILPs increase the chance of numerical issues when solving, which may lead to incorrect ILP solutions by the solver and finally to an evaluation being canceled. Such numerical issues or timeouts during ILP solving are the reason for these observed anomalies.

Following the trend observed for the dual- and quad-core systems, the greedy heuristic seems to be less effective for larger SPM sizes like the 80% setting in the lowest graph of Figure 7.6. Especially for larger utilization rates, the performance quickly degrades when compared to the fully ILP-based approaches, yet still consistently better than the evolutionary-based approach. The bus-aware ILP method performs significantly better than the other evaluated methods at utilization rates 2.2 and 2.4, while for all other utilizations, it does not improve over the reference ILP approach (with a slight exception at 2.0). Some anomalies can be seen for the fully ILP-based approaches at the 80% SPM setting, leading to a greater number of schedulable system with a larger utilization rate. As mentioned before, these anomalies are due to timeouts and numerical issues during an ILP solving phase.

7.5.5 Runtime

Figure 7.7 shows the average (arithmetic mean) runtime for each allocation optimization dependent on the number of cores per system. The runtime includes all steps of compiling, optimizing and validating the schedulability of a system. As expected, the runtime increases with the number of cores for all evaluated approaches. The reference ILP-based approach consistently requires the lowest runtime on average. The bus-aware cooperative ILP only takes slightly longer on average compared to the reference approach. Since the bus-aware cooperative ILP approach always generates and solves the bus-unaware ILP as a first step, it only requires to extend and solve the more complex ILP in case the base ILP solution did not lead to a schedulable system. This leads to only a very small increase on average over all evaluated systems. On average, the runtime when using the greedy heuristic is around twice as high compared to the bus-aware ILP approach. For small systems with only 2 cores, the runtime is on

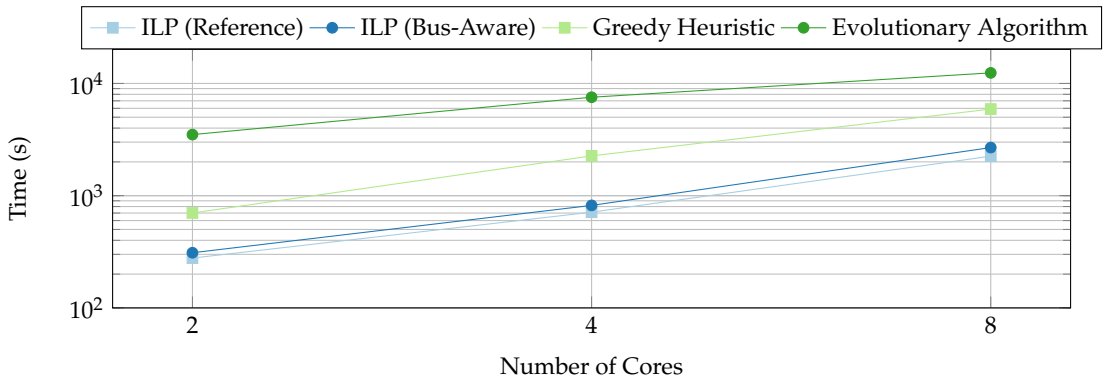


Figure 7.7. – Average runtime of the different allocation optimizations dependent on the number of cores per system.

average 11.29 times higher when using the evolutionary algorithm compared to the bus-aware ILP approach. This decreases to a factor of 9.20 for quad-core systems and 4.63 for octa-core systems.

Overall, a clear link between the required number of timing analyses for an approach and the average runtime can be drawn. As the evolutionary algorithm uses the largest number of timing analyses, the corresponding average runtime is the highest among all approaches. Since the greedy heuristic requires one timing analysis at each iteration step, it performs more analyses than the fully ILP-based ones, yet less than the evolutionary algorithm on average. Both fully ILP-based approaches require two analyses for deriving the potential gains per basic block, whereas the bus-aware cooperative ILP approach needs one additional analysis after applying the intermediate allocation.

7.5.6 Conclusion

Regarding all configurations evaluated, several conclusions can be drawn. Intuitively, the influence of bus-related timings grows with the number of cores inside a multi-core system. This directly relates to the potential of a bus-aware cooperative memory allocation approach, as the higher the influence of bus-related timings in a system is, the more important the side-effects of one core's allocation decision become to the other cores. This is noticeable when comparing the performance of the greedy heuristic and the bus-aware cooperative ILP to the reference ILP for systems with a varying number of cores. While these approaches hardly gained any improvements over the reference ILP for systems with only two cores, up to 23% more systems became schedulable compared to the reference ILP for systems with eight cores.

Furthermore, the improvements over the reference ILP are mostly seen for relatively small SPM sizes. A possible reason for this is that for larger SPM sizes, the allocation decisions for the cooperative bus-aware approaches and the reference ILP will converge in many cases, as simply most of the program parts will be placed inside the SPM. In contrast, for smaller SPM configurations it is likelier that the cooperative bus-aware approaches may allocate different program parts to the private memory than the reference ILP does. The cooperative bus-aware approaches potentially place program parts into the private memories to reduce the overall bus contention, but

do not necessarily decrease a task's WCRT. While for larger SPM sizes, these two aspects (lower bus contention and WCRT) easily coincide for most program parts (as the majority of the program fits into the private memory), this becomes less likely for smaller SPMs. This insight for the evaluated round robin-based bus arbitration is actually contradictory to the insights gained from the bus-aware SPM allocation for TDMA-based multi-core systems in Section 4.3. Here, the potential improvement by using the bus-aware ILP model increased with a larger SPM size. The reason for this are the different approaches due to the different bus arbitration techniques. As a TDMA bus arbitration enforces a complete isolation between the tasks running on different cores, each core's allocation decisions can be handled separately without the loss of any precision. Since the bus-aware ILP approach for TDMA-based multi-core systems aims at minimizing the WCRT of each task, there is a high chance of identical allocation decisions for small SPMs independent of the bus-awareness, as a few crucial program parts typically have always the largest potential (e.g., deeply nested loops). Each allocation decision in a round robin-based multi-core system in contrast may have an influence on tasks executed on other cores. Hence, an allocation minimizing a task's WCRT is easily different to an allocation minimizing the overall bus contention for the other cores. Especially for a combination of a small SPM and a system where a task can already reach its deadline without utilizing the complete SPM of its core, the allocation decision is likely to be different for the bus-aware cooperative approach and the reference ILP, only aiming at minimizing each task's WCRT.

Comparing the greedy heuristic and bus-aware cooperative ILP method, the greedy heuristic seems to work best for very small SPMs and comparably low utilization rates. The bus-aware cooperative method achieves the highest improvements for slightly larger SPMs, and also for higher utilization rates. As already seen for the bus-aware SPM allocation for TDMA-based multi-core systems, the evolutionary-based approach consistently (except for two occasions) performs the worst out of all evaluated approaches. This is mostly due to large cost of evaluating a single individual, as a timing analysis of the entire system is comparably lengthy.

8 Case Study

As a case study to present the applicability of the presented methods, a single hard real-time multi-core system is discussed and optimized in detail. All optimizations and analyses introduced in the previous chapters are applied to the system and possible combinations of optimizations are highlighted.

The following Section 8.1 introduces the system to be optimized. Subsequently, Section 8.2 showcases the results of a low-level memory allocation (as presented in Chapter 4) in case the system has a TDMA-based bus arbitration. Section 8.3 shows the results of cooperative allocation with additional shaping (as presented in Chapter 6 and Chapter 7) in case the system has a round robin-based bus arbitration.

8.1 System

For the case study, an exemplary multi-core system potentially used within the automotive domain is chosen. The used architecture is depicted in Figure 8.1. It resembles the exemplary multi-core architecture used throughout this thesis with 4 parallel cores, shared instruction and data memories, whereas each core has its own private local scratchpad memories. The bus arbitration policy is assumed to be round robin-based or TDMA. Round robin is chosen, as it is one of the most predominant bus arbitration policies of modern bus-based multi-core architectures. It is efficient due to its work-conserving nature, while still guaranteeing tight upper bounds on worst-case timings. This architecture and arbitration policy is similar[1] to the characteristics of the TriCore AURIX family [Inf14], a multi-core architecture specifically tailored towards the automotive domain. The TDMA arbitration policy is chosen as an alternative for systems, where a strict timing isolation between cores is required. For the sake of evaluation, the single cores are assumed to be ARM7TDMI-based cores with a clock frequency of 200 MHz. The SPM memories are set to a relative size of 65% of the allocated task, meaning each instruction SPM is set to 65% of the corresponding task's code size, whereas each data SPM is set to 65% of the task's data size. There is no shared data between the individual tasks. An SPM access takes a single cycle, whereas the net access latency of the shared memory (without possible interference) is 6 cycles.

Beside the actual architecture used, Figure 8.1 also shows the task allocation. Each core has one task allocated. All tasks are taken from the Embedded Microprocessor Benchmark Consortium (EEMBC) *AutoBench 1.1* suite [Emb07]. The EEMBC AutoBench suite is a collection of real-world benchmarks from the automotive domain, each performing typical functions required in this domain. In the following, the allocated benchmarks are briefly introduced:

[1] The TriCore AURIX 27x processors use a crossbar instead of a flat bus to better exploit their Harvard architecture.

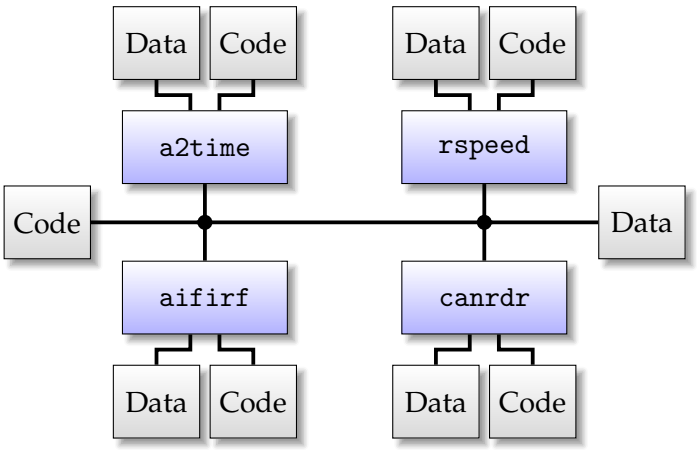


Figure 8.1. – Multi-core architecture with 4 cores with allocated tasks used in the case study.

`a2time` An engine control task. Determines the current crankshaft angle of a combustion engine and the engine speed based on a tooth pulse counter. Additionally, it controls the firing times for each cylinder.

`rspeed` A road speed calculation task. The current road speed is calculated based on counter differences. Besides, the task has to filter the input values for noise reduction and has to handle possible error states.

`aifirf` A Finite Impulse Response (FIR) filter task. Fixed-point input values are filtered by a high-pass and a low-pass FIR filter.

`canrdr` A Controller Area Network (CAN) communication task. This task simulates a CAN node and the arrival of a so-called *remote transmission request* at the node. A remote transmission request differs from a standard CAN message, as it does not contain a payload, but *asks* a specific node to send information. The simulated request frame is being evaluated by the task and eventually discarded, as the task's node is not the addressed CAN node of the request message.

These 4 benchmarks are chosen as they fulfill basic tasks required in a modern vehicle. This multi-core system could potentially be placed near the engine, as it performs necessary engine-control tasks and determines the current road speed. The filter task could be used to filter incoming or outgoing data. As most Microcontroller Units (MCUs) are connected with each other in a vehicle using field buses, the CAN task is allocated to handle this communication. It is assumed that the CAN interface is memory-mapped and can be accessed over the shared bus.

The initial WCRTs of these benchmarks with no parts allocated to the private SPMs are shown in Table 8.1, as well as their worst-case activation periods and respective deadlines. These task characteristics are chosen as follows: The activation periods of the engine control task (`a2time`) and the road speed calculation task (`rspeed`) depend on rotational speed of the crankshaft. The maximum revolutions per minute of the engine is assumed to be 9 000 RPM (similar to existing combustion engine

Table 8.1. – Characteristics of the allocated tasks.

Task	Period	Deadline	WCRT (unopt.)
`a2time`	0.11 ms (22 000 cycles)	0.11 ms (22 000 cycles)	0.340 ms (68 096 cycles)
`rspeed`	0.22 ms (44 000 cycles)	0.22 ms (44 000 cycles)	0.116 ms (23 243 cycles)
`aifirf`	1 ms (200 000 cycles)	1 ms (200 000 cycles)	1.680 ms (337 976 cycles)
`canrdr`	50 ms (10 000 000 cycles)	1 ms (200 000 cycles)	4.702 ms (940 517 cycles)

cars [Maz10]) with 60 teeth on the shaft[1] for determining the position and angular speed. In case of the engine control task, it is assumed that it is activated at every tooth of the shaft (as it needs to inject fuel at the exact right tooth position), leading to a worst-case activation period $P_{\texttt{a2time}}$:

$$P_{\texttt{a2time}} = (9000 \cdot (60\,\text{sec})^{-1} \cdot 60)^{-1} \approx 0.11\,\text{ms} \tag{8.1}$$

For the road speed calculation task, it is assumed that it does not require to be activated with every tooth passing, but only every second one. Therefore, its period is set to $2 \cdot P_{\texttt{a2time}}$. Both tasks are set to have implicit deadlines, meaning they have to finish before the next activation.

The filter task is assumed to receive incoming data from a LiDAR sensor and to process it. The LiDAR sensor has an update rate of 1 kHz, similar to existing sensors [Gar18]. This relates to an activation period of 1 ms, whereas the filter task has to finish before the next sample arrives.

The CAN task is assumed to be activated every 50 ms which is in the range of the cycle times of CAN messages in the power train of a modern car [DBBL07]. As the task is intended to check an arrived CAN message and to potentially answer to it, the deadline is set to 1 ms and not simply to an implicit deadline.

With all tasks allocated to the shared memories, all tasks except the road speed calculation miss their deadlines, hence the system is considered broken.

8.2 Low-Level Memory Allocation for TDMA

Following the requirements of the low-level memory allocation optimization presented in Chapter 4, all TDMA slots are set to a minimal length of the shared memory's net access latency of 6 cycles. In case of a multi-core system with a TDMA bus arbitration, the low-level bus-aware instruction memory allocation presented in Chapter 4 is applied. Since this optimization only decides about the allocation of the instructions, a static data memory allocation is performed beforehand. This bus-unaware WCET-oriented data memory allocation is part of the WCC framework and follows the approach initially presented by Suhendra et al. [SMRC05]. As mentioned in Chapter 4, the bus-aware low-level instruction memory allocation for TDMA-based systems can take a previous data allocation into account and derives which data accesses might access the shared bus.

After the allocation, 62.18% of the engine control task's code is placed into the private memory, 57.72% of the road speed calculation task's code, 56.24% of the filter task's code and 50.14% of the CAN task's code. The percentage of data allocated

[1] Actually only 58, as 2 teeth are removed to generate a synchronization point [YWXZ12].

Table 8.2. – Worst-case timings after the memory allocations in case of a TDMA bus.

Task	WCRT (Bus-aware Alloc.)	Schedulable?
`a2time`	23 038 cycles	✗
`rspeed`	7 965 cycles	✓
`aifirf`	65 623 cycles	✓
`canrdr`	167 668 cycles	✓

to the private memory is considerably less: 37.34% of the engine control task's data is placed into the private memory, 35.01% of the road speed calculation task's data, 2.62% of the filter task's data and 8.35% of the CAN task's data. The significantly lower percentages of data placed into the private memories are due to the lower fragmentation of the overall data of a task. While a task typically consists of dozens or hundreds of basic blocks, it often only has a few data objects, which lowers the granularity of data to be allocated to the private memory.

The worst-case timings of the tasks after applying both memory allocations are shown in Table 8.2. Overall, the engine control task's WCRT was reduced by 61%, the road speed calculation task's WCRT by 66%, the filter task's WCRT by 81% and the CAN task's WCRT by 82%. While the filter task, as well as the CAN task meet their deadline after the memory allocations applied, the engine control task still misses its deadline by around 1 000 cycles. Therefore, the instruction SPM allocation unfortunately failed to generate a schedulable system for the TDMA-based multi-core system due to the tight constraint of the engine control task.

The overall compilation, optimization and timing analysis process took 24 h and 15 min with a maximum thread count of 4. Interestingly, the vast majority of this time (24 h and 9 min) was required to solve the ILP for finding the instruction memory allocation for the engine control task, whereas all others steps were done in a total of 6 min.

As compiler-based optimizations alone could not completely repair this system, some further options exist to make the system schedulable:

- Increase the clock frequency if possible. If supported by the system, the clock frequency could be increased by 5%. This would be sufficient to repair the system (with the memory allocation determined by the low-level techniques from Chapter 4 applied).
- Ease the deadline if possible. In case the initial assumption of up to 9 000 revolutions per minute of the engine was a safe, yet exaggerated assumption, a tighter limit could be chosen. If the real upper limit of the engine's revolutions per minute is below 8 680, the system would be schedulable.
- Perform data partitioning [VSM03] beforehand to increase the efficiency of the data allocation.
- Use non-uniform slot lengths for the TDMA bus arbitration. E.g., the slot length of the engine control task's core could be increased to potentially decrease the task's WCRT.
- Upgrade the architecture. This would be the last resort in case that none of the options before are applicable. An upgrade to a more powerful architecture

Table 8.3. – Worst-case timings after the memory allocations in case of a round robin bus arbitration.

Task	WCRT (Unaware Alloc.)	WCRT (Coop. Alloc.)	Schedulable?
`a2time`	23 096 cycles	23 015 cycles	✗
`rspeed`	8 420 cycles	8 783 cycles	✓
`aifirf`	64 146 cycles	64 452 cycles	✓
`canrdr`	142 066 cycles	143 715 cycles	✓

with larger private memories would turn the system schedulable, yet would also increase costs due to the new hardware, potential re-certification steps or porting of hardware-dependent code.

8.3 Cooperative Allocation with Additional Shaping for Round Robin

In case of a multi-core system with a round robin-based bus arbitration, the combined cooperative memory allocation presented in Chapter 7 is used as a first step to optimize the tasks. Since the ILP-based variant of the cooperative memory allocation returned the best results in the majority of cases, this approach will be used for this case study as well. As a prerequisite for the cooperative memory allocation, a bus-unaware memory allocation is performed for each task. Subsequently, the actual bus-aware cooperative memory allocation is done.

After the final allocation, 62.37% of the engine control task's code is placed into the private memory, 58.07% of the road speed calculation task's code, 57.66% of the filter task's code and 59.49% of the CAN task's code. Similar to the results for the TDMA-based system of the previous section, the percentage of data allocated to the private memory is considerably less: 6.08% of the engine control task's data is placed into the private memory, 26.41% of the road speed calculation task's data, 0.3% of the filter task's data and 46.42% of the CAN task's data. Comparing these percentages to the WCET-oriented allocation for TDMA-based multi-core systems of the previous section, it is noticeable that the cooperative allocation consistently places more code into the private memory (up to 9% more), while only in one case places more data into the private memory. These differences can be caused by the different ILP model's estimated gains, which basic blocks or data object may reduce the WCRT or cause a reduced overall bus contention if a task is considered to already meet its deadline.

The timing results of both allocation steps of the cooperative allocation can be seen in Table 8.3. After the bus-unaware memory allocation, all tasks except the engine control task are meeting their deadlines. The road speed calculation task's WCRT decreased by 64%, the filter task's WCRT by 81% and the CAN task's WCRT by 85%. The engine control task's WCRT decreased by 66%, yet still violates its deadline by approximately 1 000 cycles. After performing the bus-aware cooperative memory allocation, the WCRT of all tasks except the engine control task slightly increase, while the engine control task's WCRT slightly decreases. Since the cooperative memory allocation's main goal is *not* to minimize each task's WCRT, but rather that every task's WCRT is below its respective deadline, the WCRT of a single task may easily be greater when compared to the bus-unaware memory allocation. As long as a task will meet its deadline, the cooperative memory allocation tries to allocate such program

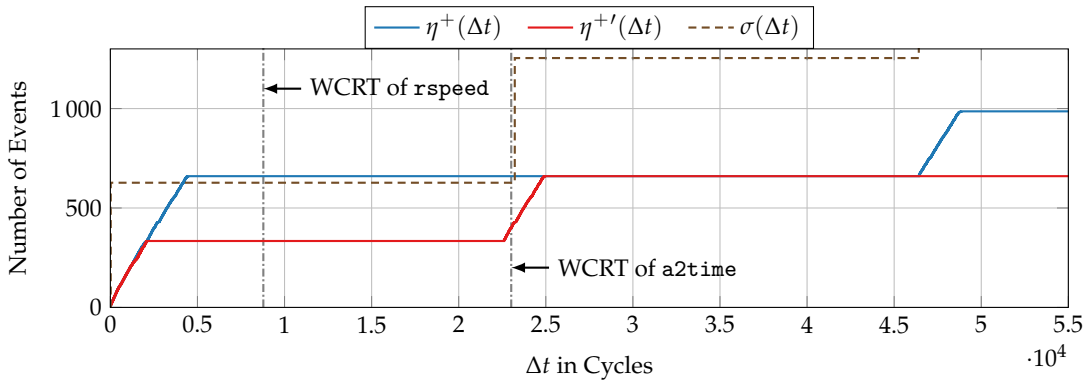


Figure 8.2. – Upper event arrival function $\eta^+(\Delta t)$ of the road speed calculation task after the cooperative memory allocation, the resulting upper event arrival function $\eta^{+\prime}(\Delta t)$ after shaping, as well as the traffic profile function $\sigma(\Delta t)$.

fragments to the private memory which potentially cause a high bus contention. This is independent from whether it will decrease the WCRT of the task which is allocated or not. Additionally, the cooperative memory allocation's ILP model integrates the timing effects of the bus, thereby increasing the precision of the estimated WCRT of the ILP model. These memory allocation decisions lead to a slightly lower WCRT of the engine control task while all other tasks still keep their respective deadlines.

The initial bus-unaware memory allocation for all four cores takes 107 s, including the two timing analysis runs for each task. The bus-aware cooperative memory allocation takes an additional 74 s, also including the timing analysis with the temporary memory allocation applied.

Since the engine control task still violates its deadline while the other tasks meet their respective deadlines by a large margin, the approach of code-inherent traffic shaping (presented in Chapter 6) is applied. The access behavior of all tasks is checked, in order to find a fitting task to be shaped to effectively reduce the maximum number of competing bus accesses for the engine control task. The maximum number of bus accesses A_{E}^{+} of the engine control task during one complete execution is 897. Given the WCRT of the engine control task of 23 015 cycles, the maximum number of bus accesses of the other tasks *during* the execution of the engine control task can be derived using each task's event arrival function:

- $\eta^{+}_{rspeed}(23\,015 \text{ cycles}) = 660$
- $\eta^{+}_{aifirf}(23\,015 \text{ cycles}) = 2\,546$
- $\eta^{+}_{canrdr}(23\,015 \text{ cycles}) = 2\,752$

To effectively reduce the number of competing bus accesses for the engine control task, a competing task's maximum number of bus accesses during a time interval of 23 015 cycles needs to be become lower than 897 to have any potential effect. This is due to the maximum bus blocking time of a round robin-based bus arbitration (cf. Equation (7.17), Page 169). Since the engine control task has at most 897 bus accesses during one complete task execution, at most 897 accesses can be blocked. E.g., reducing $\eta^{+}_{canrdr}(23\,015 \text{ cycles})$ by exemplary 1 000 accesses does not have any

Table 8.4. – Worst-case timings after the memory allocations and shaping.

Task	WCRT	Schedulable?
`a2time`	21 059 cycles	✓
`rspeed`	28 945 cycles	✓
`aifirf`	62 496 cycles	✓
`canrdr`	141 759 cycles	✓

influence on the WCRT of the engine control task, because the CAN task's bus accesses may still compete with every access of the engine control task. Due to this, the road speed calculation task is chosen for shaping, since $660 < 897$. Therefore, each access of the road speed calculation task during the WCRT of the engine control task can lead to a direct reduction of the engine control task's WCRT.

Figure 8.2 shows the upper event arrival function $\eta^+(\Delta t)$ of the road speed calculation task after the cooperative memory allocation. The traffic shaping profile $\sigma(\Delta t)$ chosen, as well as the shaped upper event arrival function $\eta^{+\prime}(\Delta t)$ are shown as well. The *full refill* shaper is chosen with the greedy heuristic as the implementation approach. The evaluation in Chapter 6 showed that this combination typically results in a very tight implementation of the shaping profile in a reasonable runtime. The shaper profile's period is set slightly above (ca. 1%) of the current WCRT of the engine control task (depicted as one of the vertical dash-dotted lines in Figure 8.2), whereas the "bucket height" is set to 95% of the current maximum accesses in this interval. This relates to a shaper profile period of 23 200 cycles and a maximum bucket height of 627 accesses.

The final worst-case timings of all tasks after the cooperative memory allocation and traffic shaping applied are shown in Table 8.4. The implementation of this shaper profile inevitably increases the WCRT of the road speed calculation task drastically. This is due to the fact that the shaper profile's period is significantly greater than the road speed calculation task's WCRT (also depicted in Figure 8.2). Therefore, certain bus accesses of the road speed calculation task have to be delayed beyond its WCRT. Yet, despite its WCRT increase, the road speed calculation task still does not violate its deadline. As planned, the shaping of the road speed calculation task leads to an overall lower bus contention, resulting in WCRT decreases for all other tasks. Most importantly, the WCRT of the engine control task is now below its deadline, making the overall system schedulable as all tasks now meet their deadlines.

The shaping process took 50 s in total, including the initial generation of the task's event arrival function's ILP model. The overall required runtime for all compilation, optimization and timing analysis steps was 20 min with a maximum thread count of 4. The additional extraction of the unshaped and shaped upper event arrival function of the road speed calculation took in total 1 h. Note that this detailed extraction is *not* necessary for the optimization or timing analyses used here, but only for the visualization.

9 Conclusion and Outlook

9.1 Summary

This thesis presented several novel approaches for optimizing multi-core systems in hard real-time systems in a WCET-aware manner and beyond. The influences of shared resources in a multi-core system not only increase the complexity of timing analyses, but also create opportunities for new optimization strategies. Whether the shared resource access behavior of a core has an influence on the other cores (as in work-conserving buses) or not (as for isolating bus strategies like TDMA) – the timing behavior of shared resources can be actively exploited to tune a multi-core system's timing behavior to the best. Applying well-known single-core WCET-directed optimizations to each task on each core in isolation can lead to good results, yet it neglects to take advantage of the full optimization potential. In order to leverage all characteristics of a multi-core system, WCET-directed optimizations need to cover all traits of these systems, including the shared resources.

To evaluate the potential of including the timing effects of shared resources into the optimization decisions and to create such novel approaches, this thesis introduced new optimization ideas specifically tailored towards multi-core systems in the domain of hard real-time systems. Additionally, it presented principles to effectively analyze multi-core architectures with a work-conserving bus arbitration in a detailed way. These techniques can be used to perform timing analyses and optimizations with detailed insights into the microarchitectural level of a multi-core architecture. Compared to previous analyses and optimizations fully integrated on the low level, these principles allow for a significantly better scalability while still maintaining detailed information compared to system-level approaches.

An ILP-based instruction memory allocation optimization for multi-core systems with a TDMA bus is presented and evaluated in Chapter 4. The optimization precisely predicts the effects of each allocation decision onto possible bus states and corresponding timing gains or penalties on processor cycle-level to find an optimal allocation for minimized worst-case timings. The evaluation showed that, despite the simplicity of a TDMA bus arbitration and its complete timing isolation between cores, significant WCRT reductions of more than 50% can be achieved compared to simply applying a single-core optimization which ignores bus-related effects.

While the bus-related timing effects of a TDMA-based multi-core system can be precisely predicted inside an optimization on a fine-grained level, this approach becomes infeasible for work-conserving bus arbitration strategies. As a foundation for optimizations and also analyses for more complex multi-core systems, potentially including a work-conserving bus arbitration, a formal description of deriving so-called *event arrival functions* from an assembly code level is presented in Chapter 5. The derivation of event arrival functions from a low level enables optimizations and analyses to work between a very detailed, yet poorly scaling microarchitectural level, and an abstract system level, which lacks a deeper insight into the system. It is shown

how lower and upper event arrival functions with numerous adjustable granularities can be described inside an ILP for tasks with arbitrary activation patterns.

How event arrival functions derived from the assembly code level can be used for optimizations is presented in Chapters 6 and 7. As a part of this, the concept of *code-inherent WCET-aware traffic shaping* is introduced in Chapter 6. Similar to traffic shapers existing in complex networks, the proposed optimization is able to shape the outgoing traffic of each core to a shared resource, yet without any additional hardware. The optimization shapes a core's traffic to any given profile by inserting delaying instructions into the allocated task, while trying to be as least invasive as possible. Therefore, two different approaches are presented, one based on evolutionary algorithms and a greedy heuristic. While this optimization can be used to reduce bus contention and to improve the worst-case timings in a multi-core system with a work-conserving bus arbitration, its application can be extended to many more cases, such as restricting the number of messages sent over a field bus to avoid a buffer overflow or to mitigate side-channel attacks. The evaluation showed that code-inherent WCET-aware traffic shaping can be successfully used to improve worst-case timings in a multi-core system, especially in case of fixed priority bus arbitration.

Whereas the previously mentioned bus-aware instruction memory allocation optimization was limited to a specific TDMA bus arbitration due to the complexity of predicting bus-related timing effects, a more general one is proposed in this thesis based on event arrival functions derived from a low level. A cooperative static instruction and data memory allocation optimization for a work-conserving bus arbitration is presented in Chapter 7. As each core's memory allocation decisions may influence tasks allocated on other cores due to the nature of a work-conserving bus arbitration, the optimization does not focus on minimizing each task's worst-case response time separately, but rather to create an overall schedulable system. This is done by not only allocating program fragments which reduce the worst-case timings of a task, but also by allocating fragments which reduce the overall bus contention to improve the timing of tasks allocated on other cores. Hence, the optimization aims at an allocation which is *cooperative* between the cores. This concept is implemented using two different approaches, a fully ILP-based one and a greedy heuristic. The evaluation showed that especially for more complex multi-core architectures, the presented approaches can improve the schedulability over a single-core directed optimization or a generic evolutionary-based memory allocation.

A case study then applies the presented optimizations to a specific quad-core system with a set of typical tasks existing in the automotive domain in Chapter 8. With the default memory allocation, all but one task fail to meet their deadlines. In case of a TDMA-based bus arbitration, the low-level memory allocation optimization presented in Chapter 4 is performed. With the memory allocation found, three of the four tasks meet their deadlines, while the fourth task's WCRT is significantly reduced, yet still slightly misses its deadline. In case of a round robin-based bus arbitration, the bus-aware cooperative memory allocation optimization from Chapter 7 is applied. Similarly, this results in three out of the four tasks meeting their deadlines after the optimization. In order to make the entire system schedulable, the principles of code-inherent traffic shaping presented in Chapter 6 is applied to one core. The bus access behavior of this task is shaped, such that it provably performs less bus accesses in a given time interval than before. This reduced number of bus accesses then decreases

the WCRTs of the other tasks, eventually turning the complete systems schedulable, as all tasks provably always meet their deadlines.

9.2 Outlook

Based on the insights and results of this thesis, several topics seem promising for future work to extend the existing principles:

Multi-Task Multi-Core Systems This thesis focused on multi-core systems with one task allocated per core. A recent publication [LOF20] by the author of this thesis and co-authors demonstrated that the principles of a schedulability-aware static instruction allocation can be extended to multi-task multi-core systems with TDMA-based bus arbitration. The presented bus-aware allocation for TDMA-based buses of this thesis had to be drastically simplified for this, as the ILP formulations became too complex to be solved in a reasonable time otherwise, even for systems with a low task and core count.

The derivation of event arrival functions presented in this thesis can be extended towards multiple tasks per core and then used as a foundation for multi-task multi-core optimizations which allow for a broader set of bus arbitration schemes. While this can be done by combining the ILP models of each task on a single core into a single ILP, it will most likely result in a very complex ILP and scale poorly with the number of tasks. The efficient derivation of an event arrival function of a multi-task system would be therefore a challenging topic for future research.

Different Interconnection Networks This thesis focused on multi-core systems with a shared bus as their interconnection network. Yet, as the number of cores inside multi-core architectures are steadily growing, future multi-core (or even many-core) architectures will increasingly adapt to interconnection networks suitable for connecting a larger number of cores, such as a NoC. With an increasing number of participants in such a network, also the average energy costs for transmitting a single packet increases, potentially even requiring to switch from traditional copper-based connections to, e.g., optical ones as presented in [BOG16]. These interconnection networks require different analysis techniques and optimizations than the presented approaches for bus-based connections, as they feature their own characteristics.

Task Dependencies This thesis covered *independent* tasks. An interesting direction for future work is the extension to include task dependencies, especially in combination with a multi-task multi-core support. Optimizations and analyses could take advantage of the known order of specific tasks to be executed due to their dependencies.

Mixed Criticality While in this thesis, all tasks of a multi-core system are considered to have hard deadlines, non-hard real-time tasks may be allocated alongside with hard real-time tasks, forming a so-called *mixed criticality* system. A possible example would be a microcontroller unit inside a modern vehicle which has to process highly

critical tasks, such as “x-by-wire”-tasks, and low critical tasks, such as infotainment-related tasks. The high critical tasks need to provably meet their deadlines under any circumstances, while the Quality of Service (QoS) of the low critical tasks should be as high as possible. Especially in a multi-core system where high and low criticality tasks perform accesses to shared resources, extending multi-core-aware optimizations to consider the different significance of the tasks would be a promising field of research.

Acronyms

BCET Best-Case Execution Time.

CAN Controller Area Network.

CFG Control-Flow Graph.

COTS Commercial Off-The-Shelf.

DMA Direct Memory Access.

EA Evolutionary Algorithm.

EEMBC Embedded Microprocessor Benchmark Consortium.

FIR Finite Impulse Response.

HLIR High-Level Intermediate Representation.

IETF Internet Engineering Task Force.

ILP Integer Linear Program.

IPET Implicit Path Enumeration Technique.

ISA Instruction Set Architecture.

LLIR Low-Level Intermediate Representation.

MBPTA Measurement-based Probabilistic Timing Analysis.

MCU Microcontroller Unit.

MTBF Mean Time Between Failure.

NC Network Calculus.

NoC Network-on-Chip.

NUMA Non-Uniform Memory Access.

pWCET Probabilistic Worst-Case Execution Time.

QoS Quality of Service.

RTC Real-Time Calculus.

SDRAM Synchronous Dynamic Random Access Memory.

SoC System-on-a-Chip.

SPM Scratchpad Memory.

SPTA Static Probabilistic Timing Analysis.

SRAM Static Random Access Memory.

TDMA Time Division Multiple Access.

UMA Uniform Memory Access.

WCC WCET-aware C Compiler.

WCEP Worst-Case Execution Path.

WCET Worst-Case Execution Time.

WCRT Worst-Case Response Time.

ILP Variables

a^+ An ILP variable representing the accumulated number of events along the sub-path contributed by a single basic block when describing an upper event arrival function. The basic block is denoted as an additional index.

a^+_{Total} An ILP variable representing the total number of accumulated events along the whole sub-path when describing an upper event arrival function.

a^- An ILP variable representing the accumulated number of events along the sub-path contributed by a single basic block when describing a lower event arrival function. The basic block is denoted as an additional index.

a^-_{Total} An ILP variable representing the total number of accumulated events along the whole sub-path when describing a lower event arrival function.

b An ILP variable representing the timing reduction factor when describing an event arrival function.

β An ILP variable describing the worst-case bus access delay of a basic block. An additional index denotes the basic block.

γ A processor's grant cycle inside a periodic bus schedule. An optional index denotes the corresponding processor.

d An ILP variable describing the bus-related timing penalty (or gain) for an explicit shared memory access. An additional index denotes the basic block, in which the access is performed.

e A binary ILP variable representing whether a basic block is used as an end of the sub-path ($e = 1$) or not. The basic block is denoted as an additional index.

f A binary ILP variable representing whether a full path was taken through the program ($f = 1$) or not ($f = 0$).

g A binary ILP variable describing whether the max. accumulated bus blocking time L is added to the WCET ($g = 1$), or whether each bus access is assumed with its worst-case delay ($g = 0$).

k An ILP variable representing the estimated number of cycles required for executing additional jump correction code. $k_{i,j}$ denotes the additional timing penalty for a jump from basic block i to j.

l An ILP variable describing the additional bus-dependent cycles required for executing jump correction code. Additional indices denote the source and target basic block.

λ An ILP variable describing the deadline violation of a task in cycles.

m^{D} An ILP variable describing the maximum number of stalled cycles to acquire a bus grant for an explicit data access depending on the bus offset interval. An additional index denotes the corresponding basic block.

m^{J} An ILP variable describing the maximum number of stalled cycles to acquire a bus grant to fetch an instruction from the shared memory during the execution of additional jump correction code depending on the bus offset interval. An additional index denotes the corresponding basic block.

μ An ILP variable describing the estimated "own" timing gain of the current task to be optimized inside the greedy heuristic of the combined cooperative memory allocation.

n_ℓ An ILP variable describing the maximum number of flows associated with a loop ℓ.

ν An ILP variable describing the estimated "cooperative" timing gain of the current task to be optimized inside the greedy heuristic of the combined cooperative memory allocation.

$\mathbf{o}$ An interval of TDMA bus offsets. An additional index describes the corresponding basic block and whether it is the incoming or outgoing offset.

$o_\downarrow$ An ILP variable representing the lower bound of a corresponding TDMA bus offset interval $\mathbf{o}$. An additional index describes the corresponding basic block and whether it is the incoming or outgoing offset.

$o_\uparrow$ An ILP variable representing the upper bound of a corresponding TDMA bus offset interval $\mathbf{o}$. An additional index describes the corresponding basic block and whether it is the incoming or outgoing offset.

p An ILP variable representing the number of times a control-flow edge is executed. The edge is denoted as an additional index.

ϕ An ILP variable used as a figure of merit to describe the bus contention caused by the current task to be optimized in the ILP-based cooperative memory allocation.

q An ILP variable representing the estimated number of bytes required for executing additional jump correction code. $q_{i,j}$ denotes the additional bytes for a jump from basic block i to j.

r An ILP variable describing the WCRT of a task. An additional index denotes the task ID.

s A binary ILP variable representing whether a control-flow edge is used as a start of the sub-path ($s = 1$) or not. The control-flow edge is denoted as an additional index.

s_ℓ A binary ILP variable, used within an ILP model to describe event arrival functions. It represents whether the sub-path starts in the loop ℓ ($s_\ell = 1$) or not ($s_\ell = 0$).

u A binary ILP variable representing whether a timing reduction is applied to the bounding basic block of the sub-path ($u = 1$) or not ($u = 0$). An additional index denotes whether the basic block is the start of the sub-path (u_s) or the end (u_e).

v A binary ILP variable representing whether a basic block is not executed along the sub-path ($v = 1$) or it is ($v = 0$). The basic block is denoted as an additional index.

w An ILP variable representing the partial accumulated WCET. An additional index describes from which basic block the partial worst-case execution path starts.

x A binary ILP variable representing the memory allocation of a single basic block. If x_i is set to 1, basic block i is moved to the fast SPM, whereas 0 represents no re-allocation.

y A binary ILP variable describing whether the corresponding task's WCRT is below its deadline ($y = 1$) or not ($y = 0$).

z^+ An ILP variable representing the accumulated timing due to executions of a single basic block when describing an upper event arrival function. An additional index denotes the basic block.

z^- An ILP variable representing the accumulated timing due to executions of a single basic block when describing a lower event arrival function. An additional index denotes the basic block.

Commonly Used Symbols and Constants

$\mathbf{A}$ The TDMA bus offset interval derived from a timing analysis. An additional index describes the corresponding basic block and to which memory it was allocated. An additional superscript denotes whether it is an incoming or outgoing offset interval.

A^{+} Maximum number of generated events during the single execution of a basic block. The corresponding basic block is denoted using an additional index.

$A_{\downarrow}$ The lower bound on a TDMA bus offset derived from a timing analysis. An additional index describes the corresponding basic block and to which memory it was allocated.

A^{+}_{E} Maximum number of generated events during one complete task execution.

A^{-} Minimum number of generated events during the single execution of a basic block. The corresponding basic block is denoted using an additional index.

$A_{\uparrow}$ The upper bound on a TDMA bus offset derived from a timing analysis. An additional index describes the corresponding basic block and to which memory it was allocated.

A^{-}_{E} Minimal number of generated events during one complete task execution.

A^{D} The constant $A^{\mathrm{D}}_{\mathtt{D0},\mathtt{B2}}$ denotes the number of times data object `D0` is accessed inside basic block `B2`.

A^{I} The constant $A^{\mathrm{I}}_{\mathtt{B2}}$ denotes the maximum number of bus accesses required to fetch the instructions of basic block `B2`.

$\mathcal{A}$ A set containing all basic blocks of a specific (sub-)path.

$\mathcal{A}_{\eta}$ A set containing all basic blocks of the sub-path from solving an ILP model describing an event arrival function.

$\mathcal{A}_{\mathrm{W}}$ A set containing all basic blocks on the program's current worst-case execution path.

B_{ℓ} Loop bound of a loop ℓ. An additional superscript denotes whether it is the upper or lower bound.

$\mathcal{B}$ Without an additional index, $\mathcal{B}$ denotes a set containing all basic blocks of a task. With an additional function as an index, it describes a set containing all basic blocks belonging to this function.

$\mathbf{C}$ Net execution time interval of a basic block, considered as a constant here. An additional index describes the associated basic block.

C^+ Net WCET of a basic block, considered as a constant here. An additional index describes the associated basic block.

C_{E}^- The minimum time required for a complete task execution which generates the maximum number of events.

C^- Net BCET of a basic block, considered as a constant here. An additional index describes the associated basic block.

C_{E}^+ The maximum time required for a complete task execution which generates the minimum number of events.

$\mathcal{C}$ A set containing all calling-edges to a given function. The function is denoted as an additional index.

D Relative deadline of a task. An additional index denotes the corresponding task.

δ^+ Upper distance function. An additional index denotes the corresponding task.

δ^- Lower distance function. An additional index denotes the corresponding task.

$\mathcal{D}$ A set containing all data objects of a task. An additional index denotes the task ID.

E The constant $E_{\mathtt{B}}$ describes the maximum possible number of executions of basic block $\mathtt{B}$, determined by the program's control-flow graph.

ϵ Threshold for the binary search used in the traffic shaping greedy heuristic.

$\mathcal{E}_\ell$ A set containing all basic blocks which are an entry of the loop ℓ.

$\mathcal{E}_\ell^{\mathrm{i}}$ A set containing all basic blocks which are an *irregular* entry of the loop ℓ.

$\mathcal{E}_\ell^{\mathrm{r}}$ A set containing all basic blocks which are a *regular* entry of the loop ℓ.

$\mathcal{F}$ Without an additional index, $\mathcal{F}$ denotes all functions of a task. With an additional index, it denotes all potential functions called from a given basic block, denoted in the index.

F_{P} Access latency of the private memory.

F_{S} Access latency of the shared memory.

G The estimated gain in terms of worst-case execution time when moving a basic block to a faster memory. An additional index denotes the basic block.

$\mathcal{G}_{\mathrm{B}}$ A set containing all basic blocks which are currently allocated to the shared memory when performing a combined cooperative memory allocation.

$\mathcal{G}_{\mathrm{D}}$ A set containing all data objects which are currently allocated to the shared memory when performing a combined cooperative memory allocation.

H A binary constant representing whether a basic block contains a potential explicit access to a shared memory ($H = 1$) or not ($H = 0$). An additional index describes the corresponding basic block.

$\mathcal{H}$ A set containing all tasks of the system allocated to cores with a higher bus priority than the core task i is executed on in case of a fixed priority-based bus arbitration.

I Minimum interarrival time between two events.

J Jitter of a task activation inside a periodic task model with jitter.

K Approximated number of cycles required for executing additional jump correction code. An additional index denotes the memory from where the jump is originating.

$\mathcal{K}$ A set containing all time interval lengths to check for validating, whether an upper event arrival function $\eta^+(\Delta t)$ adheres to a traffic profile function $\sigma(\Delta t)$ or not.

L The accumulated worst-case blocking due to bus stalling during one execution of task. An additional index denotes a task.

ℓ A single loop. An additional index may denote the loop's identifier.

$\mathcal{L}$ A set containing all loops of a task.

$\mathcal{L}_{\mathrm{H}}$ A set containing all head-controlled loops of a task.

$\mathcal{L}_{\mathrm{T}}$ A set containing all tail-controlled loops of a task.

M A sufficiently large constant.

M_{GA} Maximum amount of delay to be added to a basic block of an individual in the initial population of a genetic algorithm for traffic shaping.

$\mathcal{M}_\ell$ A set containing all basic blocks which are a member of the loop ℓ.

N_A An architecture-dependent number of cycles required between starting the execution of a memory accessing instruction and the actual memory access.

N_{C} Total number of cores in the current system.

N_{M} An integer constant defining the maximum number of basic blocks or data objects of a task allocated to the private memory in the greedy heuristic for a combined cooperative memory allocation.

N_{P} Number of individuals in an initial population of genetic algorithm.

η^+ Upper event arrival function. An additional index denotes the corresponding task.

η^+_{A} Upper event arrival function for a task's activation pattern. An additional index denotes the corresponding task.

η^- Lower event arrival function. An additional index denotes the corresponding task.

η_A^- Lower event arrival function for a task's activation pattern. An additional index denotes the corresponding task.

$\mathcal{N}_\ell$ A set containing all back edges of a loop ℓ.

P Period of a task. An additional index may denote the task.

P_M Mutation probability of a genetic algorithm.

P_T Bus period in case of a TDMA arbitrated bus.

π Worst-case access delay for a single bus access.

$\mathcal{P}$ A set containing all directly preceding basic blocks of a basic block which is denoted as an additional index.

Q_A Approximated number of bus accesses required to fetch additional jump correction code.

Q_B Approximated number of bytes required for additional jump correction code.

$Q_{S,P}$ Bus offset after performing a jump from the shared memory to the private SPM.

R A cumulative request function.

R^+ Worst-case response time of a task. An additional index denotes the task ID.

R^D The number of cycles required to perform an explicit data access to a shared memory, derived from a static timing analysis. An additional index denotes the corresponding basic block.

$\mathcal{R}$ A set containing all return-edges of a given function. The function is denoted as an additional index.

S The size of a basic block in bytes. An additional index denotes the basic block.

S_{DSPM} Total size of the data SPM. An optional superscript indicates the corresponding core ID.

S_{ISPM} Total size of the instruction SPM. An optional superscript indicates the corresponding core ID.

σ A traffic profile function.

$\mathcal{S}$ A set containing all directly succeeding basic blocks of a basic block which is denoted as an additional index.

T_ℓ A binary constant representing whether loop ℓ is tail-controlled ($T_\ell = 1$) or not ($T_\ell = 0$).

Γ A set containing all tasks of a system.

τ A task. An additional index denotes the task ID.

θ The constant θ_i describes the number of accesses contributed by the object i according to a previous analysis of the maximum number of accesses in a given time interval.

$\mathcal{T}$ A set containing all exiting blocks of a task.

U The constant $U_i^{n,m}$ describes the estimated timing gain for task τ_m when moving the object i of task τ_n to the private memory.

$\mathcal{U}$ A set containing all tasks of the system allocated to cores with a lower bus priority than the core task i is executed on in case of a fixed priority-based bus arbitration.

V A figure of merit characterizing by how much a program's event arrival function is violating a desired traffic profile.

W_{DSPM} The number of bytes currently allocated in the data SPM when performing a combined cooperative memory allocation.

W_{ISPM} The number of bytes currently allocated in the instruction SPM when performing a combined cooperative memory allocation.

X The worst-case execution count of a basic block according to the last timing analysis done. An additional index denotes the basic block.

ξ The modular distance function. $\xi_M(a, b)$ describes the positive distance between points a and b in the modular domain M.

$\mathcal{X}_\ell$ A set containing the exit blocks of a loop ℓ.

Y A binary constant representing whether a task meets its deadline according to the last timing analysis ($Y = 1$) or not ($Y = 0$). An additional index denotes the task ID.

Z The constant $Z_i^{n,m}$ describes the estimated number of effectively reduced competing bus accesses for task τ_m when moving the object i of task τ_n to the private memory.

ζ The fitness value of a single individual during an evolutionary algorithm. An additional index denotes the individual's ID.

List of Figures

List of Tables

Bibliography

[AAN11] Ernst Althaus, Sebastian Altmeyer, and Rouven Naujoks. Precise and Efficient Parametric Path Analysis. *ACM SIGPLAN Notices*, 46(5), April 2011. doi:10.1145/2016603.1967697.

[ABW09] Sebastian Altmeyer, Claire Burguière, and Reinhard Wilhelm. Computing the Maximum Blocking Time for Scheduling with Deferred Preemption. In *Proceedings of the Software Technologies for Future Dependable Distributed Systems (STFSSD)*, March 2009. doi:10.1109/STFSSD.2009.12.

[AD14] Sebastian Altmeyer and Robert I. Davis. On the correctness, optimality and precision of Static Probabilistic Timing Analysis. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, March 2014. doi:10.7873/DATE.2014.039.

[AGLS16] Peter Altenbernd, Jan Gustafsson, Björn Lisper, and Friedhelm Stappert. Early execution time-estimation through automatically generated timing models. *Real-Time Systems*, 52(6), Nov 2016. doi:10.1007/s11241-016-9250-7.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[AMWH94] Robert Arnold, Frank Mueller, David Whalley, and Marion Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, December 1994. doi:10.1109/REAL.1994.342718.

[Bar03] Sanjoy K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1), January 2003. doi:10.1023/A:1021711220939.

[BC18] Martin Becker and Samarjit Chakraborty. Optimizing Worst-Case Execution Times Using Mainstream Compilers. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, May 2018. doi:10.1145/3207719.3207739.

[BCdMS17] Armelle Bonenfant, Denis Claraz, Marianne de Michiel, and Pascal Sotin. Early WCET Prediction Using Machine Learning. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, June 2017. doi:10.4230/OASIcs.WCET.2017.5.

[Ben06] Luca Benini. *Networks on chips: technology and tools*. Elsevier Morgan Kaufmann Publishers, 2006. doi:10.1016/B978-0-12-370521-1.X5000-0.

[BLTZ03] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. Pisa—a platform and programming language independent interface for search algorithms. In *Proceedings of the International Conference on Evolutionary Multi-Criterion Optimization (EMO)*, April 2003.

[BOG16] Wolfgang Büter, Dominic Oehlert, and Alberto García-Ortiz. ER-RCA: A buffer-efficient reconfigurable optical Network-on-Chip with permanent-error recognition. In *Proceedings of the International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, June 2016. doi:10.1109/ReCoSoC.2016.7533909.

[Bor07] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the Design Automation Conference (DAC)*, June 2007. doi:10.1145/1278480.1278667.

[BRS11] R. Bourgade, C. Rochange, and P. Sainrat. Predictable bus arbitration schemes for heterogeneous time-critical workloads running on multicore processors. In *Proceedings of the International Conference on Emerging Technologies and Factory Automation (ETFA)*, September 2011. doi:10.1109/ETFA.2011.6059179.

[Bur95] Alan Burns. Preemptive Priority Based Scheduling: An Appropriate Engineering Approach. *Advances in Real-Time Systems*, March 1995.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. doi:10.1017/CBO9780511804441.

[CB02] Antoine Colin and Guillem Bernat. Scope-tree: a program representation for symbolic worst-case execution time analysis. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, June 2002. doi:10.1109/EMRTS.2002.1019185.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, January 1977. doi:10.1145/512950.512973.

[CCC+16] Sheng-Wei Cheng, Che-Wei Chang, Jian-Jia Chen, Tei-Wei Kuo, and Pi-Cheng Hsiu. Many-Core Real-Time Task Scheduling with Scratchpad Memory. *IEEE Transactions on Parallel and Distributed Systems*, 27(10), October 2016. doi:10.1109/TPDS.2016.2516519.

[CCTF13] Che-Wei Chang, Jian-Jia Chen, Tei-Wei Kuo, and Heiko Falk. Real-time partitioned scheduling on multi-core systems with local and global memories. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, January 2013. doi:10.1109/ASPDAC.2013.6509640.

[CFG+10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of Embedded Real Time Software and Systems (ERTS)*, May 2010.

[Cob19] Cobham, Wimborne Minster, UK. *GR740 Data Sheet and User's Manual*, May 2019. V2.3.

[Cob20] Cobham, Wimborne Minster, UK. *Dual-Core LEON3FT SPARC V8 Processor User Manual*, April 2020. V2.13.

[CQV+13] Francisco J. Cazorla, Eduardo Quinones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. PROARTIS: Probabilistically Analyzable Real-Time Systems. *ACM Transactions on Embedded Computing Systems*, 12(2s), May 2013. doi:10.1145/2465787.2465796.

[CR01] Carla Fabiana Chiasserini and Ramesh Rao. Improving battery performance by using traffic shaping techniques. *IEEE Journal on Selected Areas in Communications*, 19(7), July 2001. doi:10.1109/49.932705.

[CR11] Sudipta Chattopadhyay and Abhik Roychoudhury. Static Bus Schedule Aware Scratchpad Allocation in Multiprocessors. In *Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, April 2011. doi:10.1145/1967677.1967680.

[Cru91] Rene L. Cruz. A calculus for network delay, Part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1), January 1991. doi:10.1109/18.61109.

[CS06] Hugues Cassé and Pascal Sainrat. OTAWA, a framework for experimenting WCET computations. In *Proceedings of Embedded Real Time Software and Systems (ERTS)*, January 2006.

[CSBF18] Gonzalo Carvajal, Mahmoud Salem, Nirmal Benann, and Sebastian Fischmeister. Enabling Rapid Construction of Arrival Curves From Execution Traces. *IEEE Design Test*, 35(4), November 2018. doi:10.1109/MDAT.2017.2771210.

[CSH+12] Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. Measurement-Based Probabilistic Timing Analysis for Multi-path Programs. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, July 2012. doi:10.1109/ECRTS.2012.31.

[Cyp18] Cypress Semiconductor Corporation, San Jose, CA, USA. *1-Gbit (128 Mbyte)/512-Mbit (64 Mbyte)/ 256-Mbit (32 Mbyte)/128-Mbit (16 Mbyte), 3.0 V, GL-S Flash Memory*, June 2018. Rev. *R.

[DAE12] Jonas Diemer, Philip Axer, and Rolf Ernst. Compositional Performance Analysis in Python with pyCPA. In *Proceedings of the International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, July 2012.

[DAI+18] Robert Davis, Sebastian Altmeyer, Leandro S. Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. An extensible framework for multicore response time analysis. *Real-Time Systems*, 54(3), July 2018. doi:10.1007/s11241-017-9285-4.

[Dal04] William James Dally. *Principles and practices of interconnection networks*. Elsevier, 2004.

[DBBL07] Robert Davis, Alan Burns, Reinder Bril, and Johan Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35, February 2007. doi:10.1007/s11241-007-9012-7.

[DMK+18] Enrique Díaz, Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J Cazorla. Modelling multicore contention on the AURIX TM TC27x. In *Proceedings of the Design Automation Conference (DAC)*, June 2018. doi:10.1145/3195970.3196077.

[DN12a] Dakshina Dasari and Vincent Nelis. An Analysis of the Impact of Bus Contention on the WCET in Multicores. In *Proceedings of the International Conference on High Performance Computing and Communication (HPCC) & International Conference on Embedded Software and Systems (ICESS)*, January 2012. doi:10.1109/HPCC.2012.212.

[DN12b] Robert I. Davis and Nicolas Navet. Traffic Shaping to Reduce Jitter in Controller Area Network (CAN). *SIGBED Review*, 9(4), November 2012. doi:10.1145/2452537.2452544.

[DNA15] Dakshina Dasari, Vincent Nelis, and Benny Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, 52, January 2015. doi:10.1007/s11241-015-9229-9.

[DSPD19] Mickaël Dardaillon, Stefanos Skalistis, Isabelle Puaut, and Steven Derrien. Reconciling Compiler Optimizations and WCET Estimation Using Iterative Compilation. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, December 2019. doi:10.1109/RTSS46320.2019.00022.

[DZ11] Yiqiang Ding and Wei Zhang. Multicore-Aware Code Positioning to Improve Worst-Case Performance. In *Proceedings of the International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, March 2011. doi:10.1109/ISORC.2011.35.

[Emb07] Embedded Microprocessor Benchmark Consortium (EEMBC), Vancouver, WA, USA. *AutoBench 1.1 Software Benchmark Data Book*, July 2007.

[Erm03] Andreas Ermedahl. *A modular tool architecture for worst-case execution time analysis*. PhD thesis, Acta Universitatis Upsaliensis, 2003.

[FH04] Christian Ferdinand and Reinhold Heckmann. ait: Worst-case execution time prediction by static program analysis. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*, April 2004. doi:10.1109/IPDPS.2004.1303088.

[FK09] Heiko Falk and Jan C. Kleinsorge. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *Proceedings of the Design Automation Conference (DAC)*, July 2009. doi:10.1145/1629911.1630101.

[FL10] Heiko Falk and Paul Lokuciejewski. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems*, 46(2), 2010. doi:10.1007/s11241-010-9101-x.

[FS06] Heiko Falk and Martin Schwarzer. Loop Nest Splitting for WCET-Optimization and Predictability Improvement. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, October 2006. doi:10.4230/OASIcs.WCET.2006.674.

[GAF95] Raymond N. Greenwell, John Angus, and Michael Finck. Optimal mutation probability for genetic algorithms. *Mathematical and Computer Modelling*, 21(8), April 1995. doi:10.1016/0895-7177(95)00035-Z.

[Gar18] Garmin Ltd., Olathe, KS, USA. *LIDAR-Lite v3HP Operation Manual and Technical Specifications*, April 2018.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks – Past, Present and Future. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, November 2010. doi:10.4230/OASIcs.WCET.2010.136.

[GBL+00] Paolo Gai, Enrico Bini, Giuseppe Lipari, Marco Di Natale, and Luca Abeni. Architecture For A Portable Open Source Real Time Kernel Environment. In *Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, November 2000.

[GCOL16] Andrés Goens, Jeronimo Castrillon, Maximilian Odendahl, and Rainer Leupers. An optimal allocation of memory buffers for complex multicore platforms. *Journal of Systems Architecture*, 66, May 2016. doi:https://doi.org/10.1016/j.sysarc.2016.05.002.

[Gie15] Fabian Giesen. Intervals in modular arithmetic. https://fgiesen.wordpress.com/2015/09/24/intervals-in-modular-arithmetic/, 2015. [Online; accessed 22.07.2020].

[GNS+15] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Interval Analysis and Machine Arithmetic: Why Signedness Ignorance Is Bliss. *ACM Transactions on Programming Languages and Systems*, 37(1), January 2015. doi:10.1145/2651360.

[Gre93] Klaus Gresser. An Event Model for Deadline Verification of Hard Real-Time Systems. In *Proceedings of the Workshop on Real-Time Systems*, June 1993. doi:10.1109/EMWRT.1993.639067.

[GRE+01] Matthew R. Guthaus, Jeff S. Ringenberg, Daniel Ernst, Todd Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization International Workshop (WWC)*, December 2001. doi:10.1109/WWC.2001.990739.

[GSSS13] Graeme Gange, Harald Søndergaard, Peter J. Stuckey, and Peter Schachte. Solving difference constraints over modular arithmetic. In *Proceedings of the International Conference on Automated Deduction (CADE)*, June 2013. doi:10.1007/978-3-642-38574-2_15.

[GZY+15] Shouzhen Gu, Qingfeng Zhuge, Juan Yi, Jingtong Hu, and Edwin H. Sha. Optimizing Task and Data Assignment on Multi-Core Systems with Multi-Port SPMs. *IEEE Transactions on Parallel and Distributed Systems*, 26(9), September 2015. doi:10.1109/TPDS.2014.2356194.

[HAC+15] Carles Hernandez, Jaume Abella, Francisco J. Cazorla, Jan Andersson, and Andrea Gianarro. Towards making a LEON3 multicore compatible with probabilistic timing analysis. In *Proceedings of the Data Systems in Aerospace Conference (DASIA)*, May 2015.

[HAW+15] Magnus Hijorth, Martin Aberg, Nils-Johan Wessman, Jan Andersson, Remy Chevallier, Russel Forsyth, Rolad Weigand, and Luca Fossati. GR740: Rad-hard quad-core LEON4FT system-on-chip. In *Proceedings of the Data Systems in Aerospace (DASIA) conference*, May 2015.

[HHC+15] Biao Hu, Kai Huang, Gang Chen, Long Cheng, and Alois Knoll. Adaptive runtime shaping for mixed-criticality systems. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, October 2015. doi:10.1109/EMSOFT.2015.7318255.

[HHJ+05] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis - the symta/s approach. *IEE Proceedings - Computers and Digital Techniques*, 152(2), March 2005. doi:10.1049/ip-cdt:20045088.

[HHK+15] Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch, and Peter P. Puschner. The platin Tool Kit – The TCREST Approach for Compiler and WCET Integration. In *Proceedings of the Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS)*, October 2015.

[HJR16] Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling Compositionality for Multicore Timing Analysis. In *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*, October 2016. doi:10.1145/2997465.2997471.

[HJRE06] Arne Hamann, Marek Jersak, Kai Richter, and Rolf Ernst. A Framework for Modular Analysis and Exploration of Heterogeneous Embedded Systems. *Real-Time Systems*, 33(1–3), July 2006. doi:10.1007/s11241-006-6884-x.

[HZX+14] Jingtong Hu, Qingfeng Zhuge, Chun Jason Xue, Wei-Che Tseng, and Edwin H.-M. Sha. Management and Optimization for Nonvolatile Memory-Based Hybrid Scratchpad Memory on Multicore Embedded Processors. *ACM Transactions on Embedded Computing Systems*, 13(4), March 2014. doi:10.1145/2560019.

[INC20] INCHRON AG. chronVAL. https://www.inchron.com/tool-suite/chronval/, 2020. [Online; accessed 10.12.2020].

[Inf05] Informatik Centrum Dortmund e.V., Dortmund, Germany. *ICD Low Level Intermediate Representation backend infrastructure (LLIR) Developer Manual*, 2005.

[Inf07] Infineon, Neubiberg, Germany. *TC1796 User Manual*, July 2007. V2.0.

[Inf09a] Infineon, Neubiberg, Germany. *TC1797 32-Bit Single-Chip Microcontroller*, May 2009. V1.1.

[Inf09b] Informatik Centrum Dortmund e.V., Dortmund, Germany. *ICD-C Compiler framework Developer Manual*, 2009.

[Inf14] Infineon, Neubiberg, Germany. *TC27x D-Step 32-Bit Single-Chip Microcontroller*, December 2014. V2.2.

[Inf17] Informatik Centrum Dortmund e.V. ICD-CG Code Generator. http://www.icd.de/de/eingebettete-systeme/icd-c-compiler/icd-cg/, 2017. [Online; Internet Archive; accessed 08.12.2020, http://web.archive.org/web/20170510174136/http://www.icd.de/de/eingebettete-systeme/icd-c-compiler/icd-cg].

[Int18a] Integrated Silicon Solution, Inc., Milpitas, CA, USA. *128MX8, 64MX16 1Gb DDR3 SDRAM with ECC*, March 2018. Rev. B1.

[Int18b] International Organization for Standardization. ISO 26262-3: Road vehicles. Functional safety. Concept phase. Standard, Geneva, CH, December 2018.

[Jen55] A. F. Jenkinson. The frequency distribution of the annual maximum (or minimum) values of meteorological elements. *Quarterly Journal of the Royal Meteorological Society*, 81(348), April 1955. doi:10.1002/qj.49708134804.

[JHH15] Michael Jacobs, Sebastian Hahn, and Sebastian Hack. WCET Analysis for Multi-core Processors with Shared Buses and Event-driven Bus Arbitration. In *Proceedings of the International Conference on Real Time and Networks Systems (RTNS)*, November 2015. doi:10.1145/2834848.2834872.

[JHH16] Michael Jacobs, Sebastian Hahn, and Sebastian Hack. A Framework for the Derivation of WCET Analyses for Multi-core Processors. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016. doi:10.1109/ECRTS.2016.19.

[JP86] Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5), January 1986. doi:10.1093/comjnl/29.5.390.

[KBCS14] Yooseong Kim, David Broman, Jian Cai, and Aviral Shrivastava. WCET-aware dynamic code management on scratchpads for Software-Managed Multicores. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014. doi:10.1109/RTAS.2014.6926001.

[KE15] Morteza Mohajjel Kafshdooz and Alireza Ejlali. Dynamic Shared SPM Reuse for Real-Time Multicore Embedded Systems. *ACM Transactions on Architecture and Code Optimization*, 12(2), May 2015. doi:10.1145/2738051.

[Kel15] Timon Kelter. *WCET Analysis and Optimization for Multi-Core Real-Time Systems*. PhD thesis, TU Dortmund, Department of Computer Science, March 2015.

[KFM13] Jan C. Kleinsorge, Heiko Falk, and Peter Marwedel. Simple analysis of partial worst-case execution paths on general control flow graphs. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, September 2013. doi:10.1109/EMSOFT.2013.6658594.

[Kil73] Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, October 1973. doi:10.1145/512927.512945.

[Kir03] Raimund Kirner. *Extending optimising compilation to support worst-case execution time analysis*. PhD thesis, Technische Universität Wien, 2003.

[KKXL17] Michail-Alexandros Kourtis, Harilaos Koumaras, George Xilouris, and Fidel Liberal. An NFV-based video quality assessment method over 5G small cell networks. *IEEE MultiMedia*, 24(4), June 2017. doi:10.1109/MMUL.2017.265091534.

[Kle15] Jan C. Kleinsorge. *Tight Integration of Cache, Path and Task-interference Modeling for the Analysis of Hard Real-time Systems*. PhD thesis, TU Dortmund, Department of Computer Science, Dortmund, Germany, 2015.

[Knu97] Donald Ervin Knuth. *Fundamental algorithms*. The art of computer programming. Addison-Wesley, 3. edition, 1997.

[KPWF19] Daniel Kästner, Markus Pister, Simon Wegener, and Christian Ferdinand. TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, July 2019. doi:10.4230/OASIcs.WCET.2019.1.

[KSEE16] Adam Kostrzewa, Selma Saidi, Leonardo Ecco, and Rolf Ernst. Dynamic admission control for real-time networks-on-chips. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, January 2016. doi:10.1109/ASPDAC.2016.7428096.

[KT11] Pratyush Kumar and Lothar Thiele. Cool Shapers: Shaping Real-time Tasks for Improved Thermal Guarantees. In *Proceedings of the Design Automation Conference (DAC)*, June 2011. doi:10.1145/2024724.2024835.

[Lat02] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, December 2002.

[LB98] Jean-Yves Le Boudec. Application of network calculus to guaranteed service networks. *IEEE Transactions on Information Theory*, 44(3), May 1998. doi:10.1109/18.669170.

[LBT01] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001. doi:10.1007/3-540-45318-0.

[LCS92] Corinna G. Lee, P. Chow, and M. G. Stoodley. UTDSP Benchmark Suite. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html, 1992. [Online; accessed 04.05.2019].

[Leh90] John Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, December 1990. doi:10.1109/REAL.1990.128748.

[LF17] Arno Luppold and Heiko Falk. Schedulability-aware SPM Allocation for preemptive hard real-time systems with arbitrary activation patterns. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, March 2017. doi:10.23919/DATE.2017.7927149.

[LFS+07] Paul Lokuciejewski, Heiko Falk, Martin Schwarzer, Peter Marwedel, and Henrik Theiling. Influence of Procedure Cloning on WCET

Prediction. In *Proceedings of the IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, September 2007. doi:10.1145/1289816.1289852.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1), December 2007. doi:10.1016/j.scico.2007.01.014.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, December 1995. doi:10.1145/217474.217570.

[LMS15] Zhenmin Li, Avinash Malik, and Zoran Salcic. Reducing Worst Case Reaction Time of Synchronous Programs on Chip-Multiprocessors with Application-Specific TDMA Scheduling. In *Proceedings of the International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, October 2015. doi:10.1145/2822304.2822306.

[LOF20] Arno Luppold, Dominic Oehlert, and Heiko Falk. Compiling for the Worst Case: Memory Allocation for Multi-Task and Multi-Core Hard Real-Time Systems. *ACM Transactions on Embedded Computing Systems*, 19(2), March 2020. doi:10.1145/3381752.

[Lou] Louis-Noel Pouchet. PolyBench/C - The Polyhedral Benchmark Suite. http://www.cs.ucla.edu/~pouchet/software/polybench/. [Online; accessed 05.04.2019].

[LPM97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 1997. doi:10.1109/MICRO.1997.645830.

[LS99] Thomas Lundqvist and Per Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, December 1999. doi:10.1109/REAL.1999.818824.

[Lup20] Arno Luppold. *Schedulability-Oriented Code Optimization of Hard Real-Time Multitasking Systems*. PhD thesis, Hamburg University of Technology, 2020. doi:10.15480/882.2842.

[Lux20] Luxoft. Luxoft. https://www.luxoft.com, 2020. [Online; accessed 16.07.2020].

[LZ15] Yu Liu and Wei Zhang. Scratchpad Memory Architectures and Allocation Algorithms for Hard Real-Time Multicore Processors. *Journal of Computing Science and Engineering*, 9(2), June 2015. doi:10.5626/JCSE.2015.9.2.51.

[Man67] Glenn K. Manacher. Production and stabilization of real-time task schedules. *Journal of the ACM (JACM)*, 14(3), July 1967. doi:10.1145/321406.321408.

[Mar11] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Springer, 2011. doi:10.1007/978-94-007-0257-8.

[Maz10] Mazda North American Operations. 2011 MAZDA RX-8 SPECIFICATION DECK. https://insidemazda.mazdausa.com/wp-content/uploads/2017/02/2011-RX-8-Specifications.pdf, 2010. [Online; accessed 10.12.2020].

[MF20] Kateryna Muts and Heiko Falk. Multi-Criteria Function Inlining for Hard Real-Time Systems. In *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*, June 2020. doi:10.1145/3394810.3394819.

[Mic13] Microchip Technology, Chandler, AZ, USA. *1 Mbit / 2 Mbit / 4 Mbit (x8) Multi-Purpose Flash*, 06 2013. Rev. DS25023B.

[MIR13] MIRA Ltd. MISRA C: 2012 Guidelines for the use of the C language in critical systems, 2013.

[MMSH01] Gokhan Memik, William H. Mangione-Smith, and Wendong Hu. Netbench: a benchmarking suite for network processors. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, November 2001. doi:10.1109/ICCAD.2001.968595.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965. doi:10.1109/N-SSC.2006.4785860.

[Moy13] Bryon Moyer. *Real World Multicore Embedded Systems*. Newnes, 2013. doi:10.1016/B978-0-12-416018-7.00018-3.

[MS06] Milica Mitić and Mile Stojčev. A survey of three system-on-chip buses: Amba, coreconnect and wishbone. In *Proceedings of International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*, June 2006.

[Muc98] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1998.

[NSD09] Tatjana Nikolic, Mile Stojcev, and Goran Djordjevic. Cdma bus-based on-chip interconnect infrastructure. *Microelectronics Reliability*, 49(4), April 2009. doi:10.1016/j.microrel.2009.02.002.

[NXP08] NXP Semiconductors, Eindhoven, Netherlands. *MSC8122: Avoiding Arbitration Deadlock During Instruction Fetch*, October 2008. Rev. 1.

[NXP09a] NXP Semiconductors, Eindhoven, Netherlands. *LPC24XX User manual*, August 2009. Rev. 04.

[NXP09b] NXP Semiconductors, Eindhoven, Netherlands. *UM10237 LPC24XX User manual*, August 2009. Rev. 04.

[oD91] Department of Defense. MIL-HDBK-217F: RELIABILITY PREDICTION OF ELECTRONIC EQUIPMENT. Military handbook, Washington DC, USA, December 1991.

[OLF16] Dominic Oehlert, Arno Luppold, and Heiko Falk. Practical Challenges of ILP-Based SPM Allocation Optimizations. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, May 2016. doi:10.1145/2906363.2906371.

[OLF17] Dominic Oehlert, Arno Luppold, and Heiko Falk. Bus-aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, June 2017. doi:10.4230/LIPIcs.ECRTS.2017.1.

[OSF18] Dominic Oehlert, Selma Saidi, and Heiko Falk. Compiler-based Extraction of Event Arrival Functions for Real-Time Systems Analysis. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, June 2018. doi:10.4230/LIPIcs.ECRTS.2018.4.

[OSF19] Dominic Oehlert, Selma Saidi, and Heiko Falk. Code-Inherent Traffic Shaping for Hard Real-Time Systems. *ACM Transactions on Embedded Computing Systems*, 18(5s), October 2019. doi:10.1145/3358215.

[Pap81] Christos H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28(4), October 1981. doi:10.1145/322276.322287.

[PH13] David Patterson and John Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2013. doi:10.1016/B978-0-12-407726-3.00001-1.

[PL13] Linh Thi Xuan Phan and Insup Lee. Improving schedulability of fixed-priority real-time systems using shapers. In *Proceddings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013. doi:10.1109/RTAS.2013.6531094.

[PSJ+10] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, March 2010. doi:10.1109/DATE.2010.5456952.

[QLI+17] Yingjin Qian, Xi Li, Shuichi Ihara, Jürgen Kaiser, Tim Süß, and Andre Brinkmann. A Configurable Rule Based Classful Token Bucket Filter Network Request Scheduler for the Lustre File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2017. doi:10.1145/3126908.3126932.

[RAEP07] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proceedings of the International Real-Time Systems Symposium (RTSS)*, December 2007. doi:10.1109/RTSS.2007.24.

[Rap20] Rapita Systems, Inc. RapiTime. www.rapitasystems.com, 2020. [Online; accessed 08.12.2020].

[RB20] Simon Reder and Jürgen Becker. Interference-Aware Memory Allocation for Real-Time Multi-Core Systems. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2020. doi:10.1109/RTAS48715.2020.00-10.

[Rei08] Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, November 2008.

[REWSP19] Phillip Raffeck, Christian Eichler, Peter Wägemann, and Wolfgang Schröder-Preikschat. Worst-Case Energy-Consumption Analysis by Microarchitecture-Aware Timing Analysis for Device-Driven Cyber-Physical Systems. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, July 2019. doi:10.4230/OASIcs.WCET.2019.4.

[Ric04] Kai Richter. *Compositional Scheduling Analysis Using Standard Event Models. The SymTA/S Approach*. PhD thesis, Technical University Carolo-Wilhelmina of Braunschweig, 2004.

[RSDP19] Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut. Hiding Communication Delays in Contention-Free Execution for SPM-Based Multi-Core Architectures. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, July 2019. doi:10.4230/LIPIcs.ECRTS.2019.25.

[RSMN11] Kasper Revsbech, Henrik Schiøler, Tatiana K. Madsen, and Jimmy J. Nielsen. Worst-case traversal time modelling of ethernet based in-car networks using real time calculus. In Sergey Balandin, Yevgeni Koucheryavy, and Honglin Hu, editors, *Proceedings of the International Conference on Next Generation Wired/Wireless Networking (NEW2AN)*, August 2011. doi:doi.org/10.1007/978-3-642-22875-9_20.

[RTC12] RTCA. DO-178C: Software Considerations in Airborne Systems and Equipment Certification. Standard, Washington, USA, January 2012.

[RTioBKBK+09] Mehrnoush Rahmani, Ktawut Tappayuthpijarn, Author image of Benjamin Krebs Benjamin Krebs, Eckehard Steinbach, and Richard Bogenberger. Traffic shaping for resource-efficient in-vehicle communication. *IEEE Transactions on Industrial Informatics*, 5(4), May 2009. doi:10.1109/TII.2009.2019127.

[RWT+06] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, August 2006. doi:10.4230/OASIcs.WCET.2006.671.

[SC09] Richard M. Stallman and GCC Developer Community. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, 2009.

[SCF16] Mahmoud Salem, Mark Crowley, and Sebastian Fischmeister. Anomaly Detection Using Inter-Arrival Curves for Real-Time Systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016. doi:10.1109/ECRTS.2016.22.

[SDK75] Harvey M. Salkin and Cornelis A. De Kluyver. The knapsack problem: A survey. *Naval Research Logistics Quarterly*, 22(1), March 1975. doi:10.1002/nav.3800220110.

[SHRE08] Simon Schliecker, Arne Hamann, Razvan Racu, and Rolf Ernst. Formal methods for system level performance analysis and optimization. In *Proceedings of the Design Verification Conference (DVCon)*, February 2008. doi:10.24355/dbbs.084-200906150200-5.

[SIE06] Simon Schliecker, Matthias Ivers, and Rolf Ernst. Memory Access Patterns for the Analysis of MPSoCs. In *Proceedings of the IEEE North-East Workshop on Circuits and Systems (NEWCAS)*, June 2006. doi:10.1109/NEWCAS.2006.250943.

[SJ96] Thomas Schwederski and Michael Jurczyk. *Verbindungsnetze - Strukturen und Eigenschaften*. Leitfäden der Informatik. Teubner, 1996.

[SMRC05] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, December 2005. doi:10.1109/RTSS.2005.45.

[SNE09] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Response Time Analysis on Multicore ECUs With Shared Resources. *IEEE Transactions on Industrial Informatics*, 5(4), October 2009. doi:10.1109/TII.2009.2032068.

[SNE10] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, March 2010. doi:10.1109/DATE.2010.5456951.

[SPG97] Scott Shenker, Craig Partridge, and Roch Guerin. Specification of guaranteed quality of service. Internet Engineering Task Force, September 1997.

[SSIE07] Jan Staschulat, Simon Schliecker, Matthias Ivers, and Rolf Ernst. Analysis of Memory Latencies in Multi-Processor Systems. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, March 2007. doi:10.4230/OASIcs.WCET.2005.813.

[Ste10] Ingmar Jendrik Stein. *ILP-based Path Analysis on Abstract Pipeline State Graphs*. PhD thesis, Universität des Saarlandes, 2010.

[STM12] STMicroelectronics, Genf, Switzerland. *SPEAr1340 - Dual-core Cortex A9 HMI embedded MPU*, August 2012. Rev. 2.

[Stó12] Jacek Stój. Real-time communication network concept based on frequency division multiplexing. In *Proceedings of the International Conference on Computer Networks*, June 2012. doi:10.1007/978-3-642-31217-5_27.

[Str18] StreamIt Community. The StreamIt Benchmark Suite. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml, 2018. [Online; accessed 10.12.2020].

[Sym16] Symtavision GmbH. Symtavision. http://www.symtavision.com/, 2016. [Online; Internet Archive; accessed 08.12.2020, https://web.archive.org/web/20160524091516/http://www.symtavision.com/].

[Syn18] Synopsys Inc. CoMET System Engineering IDE. http://www.synopsys.com, 2018. [Online; accessed 10.12.2020].

[Tal07] Deepu Talla. An innovative HD video and digital image processor for low-cost digital entertainment products. In *Proceedings of the IEEE Hot Chips Symposium (HCS)*, August 2007. doi:10.1109/HOTCHIPS.2007.7482497.

[Tan11] Andrew S. Tanenbaum. *Computer Networks (5th edition)*. Pearson, 2011.

[TCN00] Lothas Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2000. doi:10.1109/ISCAS.2000.858698.

[Tid20] Tidorum Ltd. Bound-T Time and Stack Analyser, 2020.

[Tji93] Steve Tjiang. An olive twig. Technical report, Synopsys Inc., 1993.

[Tur37] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1), 1937. doi:10.1112/plms/s2-42.1.230.

[Tur86] Jonathan S. Turner. New directions in communications (or which way to the information age?). *IEEE Communications Magazine*, 24(10), October 1986. doi:10.1109/MCOM.1986.1092946.

[UAA18] Robert Underwood, Jason Anderson, and Amy Apon. Measuring network latency variation impacts to high performance computing application performance. In *Proceedings of the International Conference on Performance Engineering (ICPE)*, March 2018. doi:10.1145/3184407.3184427.

[VBK16] Amaury Van Bemten and Wolfgang Kellerer. Technical Report No. 201603: Network Calculus: A Comprehensive Guide. Technical report, October 2016. doi:10.13140/RG.2.2.32305.89448.

[VSM03] M. Verma, S. Steinke, and P. Marwedel. Data partitioning for maximal scratchpad usage. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, January 2003. doi:10.1109/ASPDAC.2003.1194997.

[Wan06] Ernesto Wandeler. *Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems*. PhD thesis, ETH Zürich, 2006. doi:10.3929/ethz-a-005328667.

[Was18] Wasly, Saud. *Scratchpad Memory Management For Multicore Real-Time Embedded Systems*. PhD thesis, University of Waterloo, Waterloo, Canada, 2018.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3), May 2008. doi:10.1145/1347375.1347389.

[Weg17] Simon Wegener. Towards Multicore WCET Analysis. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, June 2017. doi:10.4230/OASIcs.WCET.2017.7.

[WGSW18] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating Timing Side-Channel Leaks using Program Repair. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, July 2018. doi:10.1145/3213846.3213851.

[WMT06] Ernesto Wandeler, Alexander Maxiaguine, and Lothar Thiele. Performance analysis of greedy shapers in real-time systems. In *Proceedings of Design Automation and Test in Europe Conference (DATE)*, March 2006. doi:10.1109/DATE.2006.243801.

[WPG16] Zheng Pei Wu, Rodolfo Pellizzoni, and Danlu Guo. A composable worst case latency analysis for multi-rank DRAM devices under open row policy. *Real-Time Systems*, 52(1), April 2016. doi:10.1007/s11241-016-9253-4.

[WSW19] Jingbo Wang, Chungha Sung, and Chao Wang. Mitigating Power Side Channels during Compilation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, August 2019. doi:10.1145/3338906.3338913.

[WT06] Ernesto Wandeler and Lothar Thiele. Real-Time Calculus (RTC) Toolbox. http://www.mpa.ethz.ch/Rtctoolbox, 2006. [Online; accessed 29.07.2020].

[YWXZ12] Dong Wei Yao, Feng Wu, Huang Xie, and Yong Guang Zhang. Crank event based fuel injection and spark ignition timing control strategy for SI engines. *Advanced Materials Research*, 516, May 2012. doi:10.4028/www.scientific.net/AMR.516-517.585.

[ZCMV15] Marco Ziccardi, Alessandro Cornaglia, Enrico Mezzetti, and Tullio Vardanega. Software-enforced Interconnect Arbitration for COTS Multicores. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, July 2015. doi:10.4230/OASIcs.WCET.2015.11.

[ZLT01] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the strength Pareto evolutionary algorithm. Technical report, 2001. ETH Zürich.

[ZVSM94] Vojin Zivojnović, Juan M. Velarde, Christian Schläger, and Heinrich Meyr. DSPSTONE: A DSP-oriented Benchmarking Methodology. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT)*, October 1994.

[ZW16] Yanqi Zhou and David Wentzlaff. MITTS: Memory Inter-arrival Time Traffic Shaping. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, June 2016. doi:10.1109/ISCA.2016.53.

Appendices

Description of ILP Operators

A

Logical NOT The following Boolean function is given:

$$y = \overline{x} \tag{A.1}$$

where $\overline{x}$ implies the Boolean NOT operator applied to the binary variable x. This is easily described using the following constraint:

$$y = 1 - x \tag{A.2}$$

Logical OR The following Boolean function with N binary variables is given:

$$y = x_0 \vee x_1 \vee x_2 \vee \cdots \vee x_{N-1} \tag{A.3}$$

where $\vee$ is the logical OR operator and the result variable y is bound to binary values. Independent from the number of operands, this can efficiently be described using the three following constraints and a helper variable s:

$$s = x_0 + x_1 + x_2 + \cdots + x_{N-1} \tag{A.4}$$
$$0 \leq s - y \tag{A.5}$$
$$s \leq y \cdot N \tag{A.6}$$

Equation (A.4) forces the integer variable s to be equal to the sum of all binary variables. In case all variables x_0 to x_{N-1} are equal to 0, Equation (A.5) forces the result variable y to 0 as well. Similarly, Equation (A.6) enforces y to be 1 in case at least one of the x_0 to x_{N-1} variables is equal to 1.

Logical AND The following Boolean function with N binary variables is given:

$$y = x_0 \wedge x_1 \wedge x_2 \wedge \cdots \wedge x_{N-1} \tag{A.7}$$

where $\wedge$ is the logical AND operator and the result variable y is bound to binary values. Independent of the number of operands, this can efficiently be described using the three following constraints and a helper variable s:

$$s = x_0 + x_1 + x_2 + \cdots + x_{N-1} \tag{A.8}$$
$$N \leq s + (1 - y) \cdot N \tag{A.9}$$
$$N \geq s - y \cdot N + 1 \tag{A.10}$$

Equation (A.4) forces the integer variable s to be equal to the sum of all binary "input" variables x_i. In case at least one of the "input" variables x_i is equal to 0,

Equation (A.9) forces the result variable y to 0 as well. Similarly, Equation (A.6) enforces y to be 1 in case all "input" variables are equal to 1.

Modulo Function The definition of a modulo function described by Knuth [Knu97] is used:

$$m = x \bmod Y = x - \left\lfloor \frac{x}{Y} \right\rfloor \cdot Y \text{ , if } Y \neq 0 \tag{A.11}$$

where x is an integer variable and Y is an integer constant. The resulting value m has the same sign as the divisor Y. Equation (A.11) is re-formulated as follows:

$$x < (q+1) \cdot Y \tag{A.12}$$
$$x \geq Y \cdot q \tag{A.13}$$
$$m = x - q \cdot Y \tag{A.14}$$

Equations (A.12) and (A.13) implement the floored division by q holding the result of $\lfloor \frac{x}{Y} \rfloor$. Y is assumed to be always non-zero. The variable m is set to the modulo result by Equation (A.14).

Conditional Assignment of ILP Variables The following conditional assignment is given:

$$a = \begin{cases} u & \text{if } c = 1, \\ v & \text{else.} \end{cases} \tag{A.15}$$

with a, u, v and c being ILP variables, whereas the condition variable c is restricted to binary values. Equation (A.15) is expressed as a set of inequations to formulate *if-then-else* structures inside an ILP model.

$$a \geq u - (1-c) \cdot M \tag{A.16}$$
$$a \leq u + (1-c) \cdot M \tag{A.17}$$
$$a \geq v - c \cdot M \tag{A.18}$$
$$a \leq v + c \cdot M \tag{A.19}$$

Here, M is a sufficiently large constant. Equations (A.16) and (A.17) fix variable a to the value of u in case $c = 1$, otherwise a is only constrained to satisfy $u - M \leq a \leq u + M$. Analogously, Equations (A.18) and (A.19) force a to the value of v in case $c = 0$, while in the opposite case, a is solely constrained to $v - M \leq a \leq v + M$.

Min/Max Function Given are the following two functions:

$$\max(x, y) \tag{A.20}$$
$$\min(x, y) \tag{A.21}$$

with x and y being ILP variables. In order to express these two functions in terms of ILP constraints, an ILP variable c restricted to binary values is created, used as a condition.

$$y \leq x + c \cdot M \tag{A.22}$$
$$x \leq y + (1 - c) \cdot M \tag{A.23}$$

M is a sufficiently large constant. Equations (A.22) and (A.23) set c to 1 in case $y > x$, otherwise c is forced to 0. Based upon this, the functions $\max(x, y)$ and $\min(x, y)$ can be represented using the following case statement:

$$\max(x, y) = \begin{cases} y & \text{if } c = 1, \\ x & \text{else.} \end{cases} \tag{A.24}$$

$$\min(x, y) = \begin{cases} x & \text{if } c = 1, \\ y & \text{else.} \end{cases} \tag{A.25}$$

The case statement used in equations Equations (A.24) and (A.25) can be represented using the conditional assignment of ILP variables as shown before.

Ceiling Division The following equation is given:

$$q = \left\lceil \frac{x}{Y} \right\rceil \tag{A.26}$$

where x is an integer variable and Y is an integer constant. This equation can be re-formulated to the following inequations:

$$x \leq q \cdot Y \tag{A.27}$$
$$x > (q - 1) \cdot Y \tag{A.28}$$

B Modulo Interval Inclusion Condition

Proposition B.1. *An interval* $\tilde{\mathbf{o}} = [\tilde{o_\downarrow}, \tilde{o_\uparrow}]_{P_\mathrm{T}}$ *completely includes another interval* $\mathbf{o} = [o_\downarrow, o_\uparrow]_{P_\mathrm{T}}$ *if* $\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, \tilde{o_\uparrow}) \geq \min\left(\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, o_\downarrow) + \xi_{P_\mathrm{T}}(o_\downarrow, o_\uparrow), P_\mathrm{T} - 1\right)$ *holds.*

Proof. All possible cases and their implications on the proposed condition are evaluated. If the interval $\tilde{\mathbf{o}}$ completely includes another interval $\mathbf{o}$ (see Figure 4.8, Page 47), the following conditions are met:

$$\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, o_\downarrow) \leq \xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, \tilde{o_\uparrow}) \tag{B.1}$$
$$\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, o_\uparrow) \leq \xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, \tilde{o_\uparrow}) \tag{B.2}$$

Equation (B.1) describes that the offset $o_\downarrow$ has to be inside the interval $\tilde{\mathbf{o}}$, whereas Equation (B.2) describes that $o_\uparrow$ has to be inside the interval as well. Equation (B.2) can be reformulated by describing $\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, o_\uparrow)$ with the interval $\mathbf{o}$'s length:

$$\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, o_\downarrow) + \xi_{P_\mathrm{T}}(o_\downarrow, o_\uparrow) \leq \xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, \tilde{o_\uparrow}) \tag{B.3}$$

As Equation (B.3) resembles the proposed condition and Equation (B.1) is always kept if Equation (B.3) holds, the proposed condition is satisfied if an interval $\tilde{\mathbf{o}}$ completely includes another interval $\mathbf{o}$.

There are 4 different cases, for which the interval $\tilde{\mathbf{o}}$ does not completely include the interval $\mathbf{o}$.

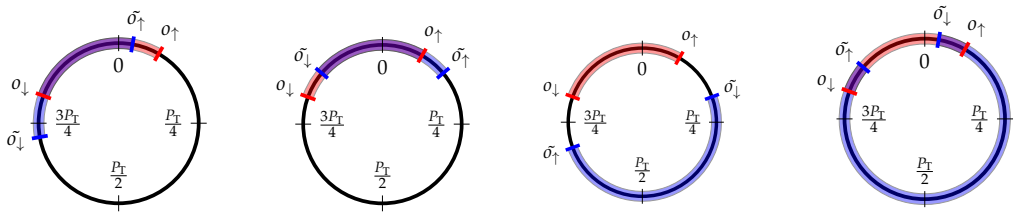
1. $o_\downarrow$ is in $\tilde{\mathbf{o}}$, but $o_\uparrow$ not.
2. $o_\uparrow$ is in $\tilde{\mathbf{o}}$, but $o_\downarrow$ not.
3. Neither $o_\downarrow$, nor $o_\uparrow$ are in $\tilde{\mathbf{o}}$.
4. Both, $o_\downarrow$ and $o_\uparrow$, are in $\tilde{\mathbf{o}}$, but not the complete interval $\mathbf{o}$.

Case 1 This case is depicted in Figure B.1(a) where the interval $\tilde{\mathbf{o}}$ includes $o_\downarrow$, but not $o_\uparrow$. For this case, the following two conditions need to be fulfilled:

$$\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, o_\downarrow) \leq \xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, \tilde{o_\uparrow}) \tag{B.4}$$
$$\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, o_\uparrow) > \xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, \tilde{o_\uparrow}) \tag{B.5}$$

Equation (B.4) describes that the offset $o_\downarrow$ is included in the interval $\tilde{\mathbf{o}}$, whereas Equation (B.5) forces the offset $o_\uparrow$ to be outside of $\tilde{\mathbf{o}}$. Equation (B.5) can be reformulated by replacing $\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, o_\uparrow)$ as follows:

$$\xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, o_\downarrow) + \xi_{P_\mathrm{T}}(o_\downarrow, o_\uparrow) > \xi_{P_\mathrm{T}}(\tilde{o_\downarrow}, \tilde{o_\uparrow}) \tag{B.6}$$



(a) Case 1 **(b)** Case 2 **(c)** Case 3 **(d)** Case 4

Figure B.1. – Different cases for which an interval $\tilde{\mathbf{o}}$ does not completely include another interval $\mathbf{o}$.

Equation (B.6) contradicts the condition from Proposition B.1, as the relational operator is flipped and the left-hand side of Equation (B.6) equals the first part of the min-term of the proposed condition. There cannot exist a case where the interval $\tilde{\mathbf{o}}$ includes $o_\downarrow$ but not $o_\uparrow$ and the proposed condition holds.

Case 2 This case is depicted in Figure B.1(b) where the interval $\tilde{\mathbf{o}}$ includes $o_\uparrow$, but not $o_\downarrow$. For this case, the following two conditions need to be fulfilled:

$$\xi_{P_T}(\tilde{o_\downarrow}, o_\uparrow) \leq \xi_{P_T}(\tilde{o_\downarrow}, \tilde{o_\uparrow}) \tag{B.7}$$
$$\xi_{P_T}(\tilde{o_\downarrow}, o_\downarrow) > \xi_{P_T}(\tilde{o_\downarrow}, \tilde{o_\uparrow}) \tag{B.8}$$

Equation (B.7) describes that the offset $o_\uparrow$ is included in the interval $\tilde{\mathbf{o}}$, whereas Equation (B.8) forces the offset $o_\downarrow$ to be outside of $\tilde{\mathbf{o}}$. As Equation (B.8) contradicts the proposed condition, there cannot be a case where the interval $\tilde{\mathbf{o}}$ includes $o_\uparrow$, but not $o_\downarrow$ and the condition holds true.

Case 3 This case is depicted in Figure B.1(c) where the interval $\tilde{\mathbf{o}}$ does not include $o_\uparrow$ and neither $o_\downarrow$. For this case, the following two conditions need to be fulfilled:

$$\xi_{P_T}(\tilde{o_\downarrow}, o_\uparrow) > \xi_{P_T}(\tilde{o_\downarrow}, \tilde{o_\uparrow}) \tag{B.9}$$
$$\xi_{P_T}(\tilde{o_\downarrow}, o_\downarrow) > \xi_{P_T}(\tilde{o_\downarrow}, \tilde{o_\uparrow}) \tag{B.10}$$

Equations (B.9) and (B.10) describe that none of the two offsets are inside the interval $\tilde{\mathbf{o}}$. As Equation (B.10) let alone already contradicts the proposed condition, there cannot be a case where the interval $\tilde{\mathbf{o}}$ includes $o_\uparrow$, but not $o_\downarrow$ and the condition holds true.

Case 4 This case is depicted in Figure B.1(d) where the interval $\tilde{\mathbf{o}}$ includes $o_\downarrow$ and $o_\uparrow$, but not the entire interval $\mathbf{o}$. For this case, the following condition needs to be fulfilled:

$$\xi_{P_T}(\tilde{o_\downarrow}, o_\downarrow) + \xi_{P_T}(o_\downarrow, o_\uparrow) \geq P_T - 1 \tag{B.11}$$

As a consequence of Equation (B.11), the proposed condition only holds true if $\xi_{P_T}(\tilde{o_\downarrow}, \tilde{o_\uparrow})$ is greater than or equal to $P_T - 1$ due to the min-term. If the interval $\tilde{\mathbf{o}}$ has a length of $P_T - 1$, it includes all possible other intervals, since it includes all possible

offsets. Hence, there cannot be a case where the interval $\tilde{\mathbf{o}}$ includes $o_{\downarrow}$ and $o_{\uparrow}$, but not the entire interval $\mathbf{o}$ and the proposed condition holds true. □

C

Evolutionary Algorithm for a Static Instruction Memory Allocation

This chapter briefly describes the evolutionary algorithm used for a WCRT-oriented static instruction memory allocation first presented during the Euromicro Conference on Real-Time Systems 2017 [OLF17]. A more detailed description, as well as an extension to multi-task systems, is given by Luppold [Lup20], who is also the main author of this optimization.

Genome Composition Each individual's genome is represented using N_C lists, where each list contains the binary allocations for all basic blocks of a task allocated to a core.

Initial Population For the initial population, one individual is created for which all basic blocks of all cores remain in the default shared memory. This individual should represent the initial, unoptimized state where everything resides in the shared memory. For all other individuals, all basic blocks of all cores are set to be placed in the private memory. Since this is most likely not possible due to insufficient private memory space, the repair function is initially called for each of these individuals.

Recombination For recombination, a simple single point crossover is used. A single random crossover point is chosen among all basic block decisions of all cores. The newly created individual's genome will consist of the basic block decisions of the first parent until the crossover point and of the decisions of the second parent for the rest.

Mutation One of the cores is randomly selected for mutation and a maximum number M of mutations is drawn from the interval $[0, |\mathcal{B}| - 1]$, where $|\mathcal{B}|$ is the number of basic blocks of the task allocated to the selected core. Then, a random basic block is drawn M times and the corresponding decision variable is flipped with a user-given probability P_M.

Repair Function An invalid solution due to too many basic blocks assigned to the private memory is repaired using a repair function. This function places random basic blocks back into the shared memory until the private memory is not overfull anymore. Due to the new memory allocation, a jump correction has to be performed afterwards, which may add some instructions to the private memory again. This leads to an iterative behavior, as the repair function may have to remove then again some basic blocks.

Fitness Function The fitness of an individual is set to the sum of the single tasks' WCRTs. Therefore, the fitness function tries to *minimize* the accumulated WCRTs of a system.

D

Determining the Maximum Execution Count of a Basic Block

To determine the maximum possible execution count of a given basic block, a slightly modified IPET model can be used. An example for this ILP model is briefly shown in the following. For this purpose, the exemplary CFG depicted in Figure D.1 is used. Following the principles of the basic IPET model as proposed by Li and Malik [LM95], the following constraints are used to implicitly describe all possible paths of the given CFG:

$$1 = p_{\texttt{B0,B1}} + p_{\texttt{B0,B3}} \tag{D.1}$$

$$p_{\texttt{B0,B1}} = p_{\texttt{B1,B2}} \tag{D.2}$$

$$p_{\texttt{B0,B3}} + p_{\texttt{B4,B3}} = p_{\texttt{B3,B5}} \tag{D.3}$$

$$p_{\texttt{B6,B4}} + p_{\texttt{B7,B4}} = p_{\texttt{B4,B2}} + p_{\texttt{B4,B3}} \tag{D.4}$$

$$p_{\texttt{B3,B5}} = p_{\texttt{B5,B6}} + p_{\texttt{B5,B7}} \tag{D.5}$$

$$p_{\texttt{B5,B6}} = p_{\texttt{B6,B4}} \tag{D.6}$$

$$p_{\texttt{B5,B7}} = p_{\texttt{B7,B4}} \tag{D.7}$$

$$p_{\texttt{B4,B3}} = 9 \cdot p_{\texttt{B0,B3}} \tag{D.8}$$

The variable $p_{\texttt{B0,B1}}$ describes how many times the control-flow edge from basic block `B0` to `B1` is taken. To determine how many times a given basic block `B` can be executed at most during a complete execution of a program, the ILP's objective is set to maximize the number of executions of the given block `B`:

$$\max : \sum_{i \in \mathcal{P}_{\texttt{B}}} p_{i,\texttt{B}} \tag{D.9}$$

The set $\mathcal{P}_{\texttt{B}}$ contains all direct predecessors of basic block `B`. By solving the ILP, the maximum possible execution count of the given basic block `B` is returned as the objective value.

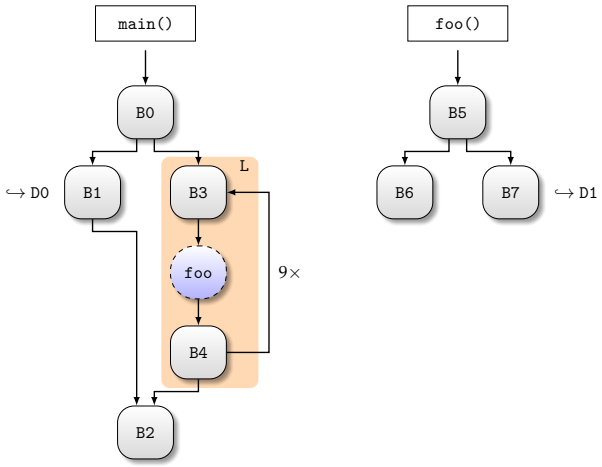


Figure D.1. – Exemplary control-flow graph.

Overview of Used Benchmarks

E

E.1 Benchmarks for Multi-Core-aware Static Instruction Allocation

Table E.1. – Benchmarks and their corresponding IDs used in the multi-core-aware static instruction allocation for TDMA-based systems.

ID	Name	ID	Name
0	codecs_codrle1 (misc)	41	jpeg (UTDSP)
1	codecs_dcodhuff (misc)	42	codecs_dcodrle1 (misc)
2	selection_sort (misc)	43	mult_4_4 (UTDSP)
3	trellis (UTDSP)	44	g723_encode (misc)
4	crc (MRTC)	45	bsort100 (MRTC)
5	lmsfir_32_64 (UTDSP)	46	iir_1_1 (UTDSP)
6	nsichneu (MRTC)	47	epic (MediaBench)
7	fdct (MRTC)	48	insertsort (MRTC)
8	lcdnum (MRTC)	49	expint (MRTC)
9	fft_1024 (UTDSP)	50	edge_detect (UTDSP)
10	qmf_receive (UTDSP)	51	utdspAdpcm (UTDSP)
11	g721_encode (misc)	52	latnrm_8_1 (UTDSP)
12	st (MRTC)	53	mpeg2 (MediaBench)
13	qurt (MRTC)	54	janne_complex (MRTC)
14	hamming_window (misc)	55	h264dec_ldecode_macroblock (MediaBench)
15	minver (MRTC)	56	fir (MRTC)
16	fac (MRTC)	57	statemate (MRTC)
17	compress (MRTC)	58	qmf_transmit (UTDSP)
18	fir_32_1 (UTDSP)	59	mult_10_10 (UTDSP)
19	cjpeg_jpeg6b_wrbmp (MediaBench)	60	adpcm (MRTC)
20	edn (MRTC)	61	matmult (MRTC)
21	latnrm_32_64 (UTDSP)	62	pm (misc)
22	gsm_encode (MediaBench)	63	gsm (MediaBench)
23	sqrt (MRTC)	64	cover (MRTC)
24	searchmultiarray (misc)	65	fft_256 (UTDSP)
25	iir_4_64 (UTDSP)	66	jfdctint (MRTC)
26	cnt (MRTC)	67	select (MRTC)
27	h263 (misc)	68	fmref (StreamIt)
28	codecs_codhuff (misc)	69	lms (MRTC)
29	fibcall (MRTC)	70	prime (MRTC)
30	spectral (UTDSP)	71	ud (MRTC)
31	ndes (MRTC)	72	test3 (misc)
32	ns (MRTC)	73	filterbank (StreamIt)
33	petrinet (MRTC)	74	anagram (misc)
34	histogram (UTDSP)	75	ammunition (misc)
35	cjpeg_jpeg6b_transupp (MediaBench)	76	fir_256_64 (UTDSP)
36	lmsfir_8_1 (UTDSP)	77	fft1 (MRTC)
37	lpc (UTDSP)	78	audiobeam (StreamIt)
38	gsm_decode (MediaBench)	79	ludcmp (MRTC)
39	h264dec_ldecode_block (MediaBench)	80	utdspCompress (UTDSP)
40	bs (MRTC)		

E.2 Benchmark Sets for the Event Arrival Function-based Evaluations

Table E.2. – Benchmark sets for dual-core systems. The order of the benchmarks inside a set denote the assigned core, i.e., the first benchmark is assigned to core 0, the second to core 1, etc.

Set	Benchmark Name	Set	Benchmark Name
1	g721_encode (misc)	44	ud (MRTC)
1	g721.marcuslee_encoder (UTDSP)	44	fft_1024 (UTDSP)
2	g721.marcuslee_encoder (UTDSP)	45	fft_1024 (UTDSP)
2	qurt (MRTC)	45	lms (MRTC)
3	qurt (MRTC)	46	lms (MRTC)
3	test3 (misc)	46	gsm_encode (MediaBench)
4	test3 (misc)	47	gsm_encode (MediaBench)
4	fir_256_64 (UTDSP)	47	audiobeam (StreamIt)
5	fir_256_64 (UTDSP)	48	audiobeam (StreamIt)
5	fft1 (MRTC)	48	ammunition (misc)
6	fft1 (MRTC)	49	ammunition (misc)
6	cnt (MRTC)	49	spectral (UTDSP)
7	cnt (MRTC)	50	spectral (UTDSP)
7	iir_1_1 (UTDSP)	50	v32.modem_cnoise (UTDSP)
8	iir_1_1 (UTDSP)	51	v32.modem_cnoise (UTDSP)
8	h264dec_ldecode_block (MediaBench)	51	g721.marcuslee_decoder (UTDSP)
9	h264dec_ldecode_block (MediaBench)	52	g721.marcuslee_decoder (UTDSP)
9	fir_32_1 (UTDSP)	52	ndes (MRTC)
10	fir_32_1 (UTDSP)	53	ndes (MRTC)
10	lpc (UTDSP)	53	cjpeg_jpeg6b_wrbmp (MediaBench)
11	lpc (UTDSP)	54	cjpeg_jpeg6b_wrbmp (MediaBench)
11	v32.modem_achop (UTDSP)	54	fdct (MRTC)
12	v32.modem_achop (UTDSP)	55	fdct (MRTC)
12	gsm (MediaBench)	55	cjpeg_jpeg6b_transupp (MediaBench)
13	gsm (MediaBench)	56	cjpeg_jpeg6b_transupp (MediaBench)
13	select (MRTC)	56	insertsort (MRTC)
14	select (MRTC)	57	insertsort (MRTC)
14	prime (MRTC)	57	jfdctint (MRTC)
15	prime (MRTC)	58	jfdctint (MRTC)
15	bs (MRTC)	58	qmf_receive (UTDSP)
16	bs (MRTC)	59	qmf_receive (UTDSP)
16	h264dec_ldecode_macroblock (MediaBench)	59	pm (misc)
17	h264dec_ldecode_macroblock (MediaBench)	60	pm (misc)
17	janne_complex (MRTC)	60	selection_sort (misc)
18	janne_complex (MRTC)	61	selection_sort (misc)
18	v32.modem_bencode (UTDSP)	61	fft_256 (UTDSP)
19	v32.modem_bencode (UTDSP)	62	fft_256 (UTDSP)
19	adpcm (MRTC)	62	lmsfir_8_1 (UTDSP)
20	adpcm (MRTC)	63	lmsfir_8_1 (UTDSP)
20	utdspAdpcm (UTDSP)	63	codecs_codrle1 (misc)
21	utdspAdpcm (UTDSP)	64	codecs_codrle1 (misc)
21	searchmultiarray (misc)	64	expint (MRTC)
22	searchmultiarray (misc)	65	expint (MRTC)
22	v32.modem_ddecode (UTDSP)	65	epic (MediaBench)
23	v32.modem_ddecode (UTDSP)	66	epic (MediaBench)
23	v32.modem_eglue (UTDSP)	66	bsort100 (MRTC)
24	v32.modem_eglue (UTDSP)	67	bsort100 (MRTC)
24	compress (MRTC)	67	codecs_dcodhuff (misc)

Set	Benchmark Name	Set	Benchmark Name
25	compress (MRTC)	68	codecs_dcodhuff (misc)
25	filterbank (StreamIt)	68	lmsfir_32_64 (UTDSP)
26	filterbank (StreamIt)	69	lmsfir_32_64 (UTDSP)
26	histogram (UTDSP)	69	latnrm_8_1 (UTDSP)
27	histogram (UTDSP)	70	latnrm_8_1 (UTDSP)
27	codecs_dcodrle1 (misc)	70	trellis (UTDSP)
28	codecs_dcodrle1 (misc)	71	trellis (UTDSP)
28	h263 (misc)	71	crc (MRTC)
29	h263 (misc)	72	crc (MRTC)
29	qsort-exam (MRTC)	72	sqrt (MRTC)
30	qsort-exam (MRTC)	73	sqrt (MRTC)
30	mult_10_10 (UTDSP)	73	fir (MRTC)
31	mult_10_10 (UTDSP)	74	fir (MRTC)
31	mult_4_4 (UTDSP)	74	lcdnum (MRTC)
32	mult_4_4 (UTDSP)	75	lcdnum (MRTC)
32	iir_4_64 (UTDSP)	75	edge_detect (UTDSP)
33	iir_4_64 (UTDSP)	76	edge_detect (UTDSP)
33	ns (MRTC)	76	statemate (MRTC)
34	ns (MRTC)	77	statemate (MRTC)
34	st (MRTC)	77	petrinet (MRTC)
35	st (MRTC)	78	petrinet (MRTC)
35	utdspCompress (UTDSP)	78	cover (MRTC)
36	utdspCompress (UTDSP)	79	cover (MRTC)
36	edn (MRTC)	79	fac (MRTC)
37	edn (MRTC)	80	fac (MRTC)
37	g723_encode (misc)	80	qmf_transmit (UTDSP)
38	g723_encode (misc)	81	qmf_transmit (UTDSP)
38	jpeg (UTDSP)	81	hamming_window (misc)
39	jpeg (UTDSP)	82	hamming_window (misc)
39	latnrm_32_64 (UTDSP)	82	ludcmp (MRTC)
40	latnrm_32_64 (UTDSP)	83	ludcmp (MRTC)
40	mpeg2 (MediaBench)	83	gsm_decode (MediaBench)
41	mpeg2 (MediaBench)	84	gsm_decode (MediaBench)
41	minver (MRTC)	84	fibcall (MRTC)
42	minver (MRTC)	85	fibcall (MRTC)
42	matmult (MRTC)	85	nsichneu (MRTC)
43	matmult (MRTC)	86	nsichneu (MRTC)
43	ud (MRTC)	86	g721_encode (misc)

Table E.3. – Benchmark sets for quad-core systems. The order of the benchmarks inside a set denote the assigned core, i.e., the first benchmark is assigned to core 0, the second to core 1, etc.

Set	Benchmark Name	Set	Benchmark Name
1	fft1 (MRTC)	44	latnrm_32_64 (UTDSP)
1	lmsfir_32_64 (UTDSP)	44	codecs_dcodrle1 (misc)
1	nsichneu (MRTC)	44	epic (MediaBench)
1	v32.modem_ddecode (UTDSP)	44	cjpeg_jpeg6b_wrbmp (MediaBench)
2	lmsfir_32_64 (UTDSP)	45	codecs_dcodrle1 (misc)
2	nsichneu (MRTC)	45	epic (MediaBench)
2	v32.modem_ddecode (UTDSP)	45	cjpeg_jpeg6b_wrbmp (MediaBench)
2	iir_4_64 (UTDSP)	45	v32.modem_cnoise (UTDSP)
3	nsichneu (MRTC)	46	epic (MediaBench)
3	v32.modem_ddecode (UTDSP)	46	cjpeg_jpeg6b_wrbmp (MediaBench)
3	iir_4_64 (UTDSP)	46	v32.modem_cnoise (UTDSP)
3	select (MRTC)	46	audiobeam (StreamIt)
4	v32.modem_ddecode (UTDSP)	47	cjpeg_jpeg6b_wrbmp (MediaBench)
4	iir_4_64 (UTDSP)	47	v32.modem_cnoise (UTDSP)
4	select (MRTC)	47	audiobeam (StreamIt)
4	insertsort (MRTC)	47	crc (MRTC)
5	iir_4_64 (UTDSP)	48	v32.modem_cnoise (UTDSP)
5	select (MRTC)	48	audiobeam (StreamIt)
5	insertsort (MRTC)	48	crc (MRTC)
5	bs (MRTC)	48	histogram (UTDSP)

Set	Benchmark Name	Set	Benchmark Name
6	select (MRTC)	49	audiobeam (StreamIt)
6	insertsort (MRTC)	49	crc (MRTC)
6	bs (MRTC)	49	histogram (UTDSP)
6	selection_sort (misc)	49	cjpeg_jpeg6b_transupp (MediaBench)
7	insertsort (MRTC)	50	crc (MRTC)
7	bs (MRTC)	50	histogram (UTDSP)
7	selection_sort (misc)	50	cjpeg_jpeg6b_transupp (MediaBench)
7	mult_4_4 (UTDSP)	50	trellis (UTDSP)
8	bs (MRTC)	51	histogram (UTDSP)
8	selection_sort (misc)	51	cjpeg_jpeg6b_transupp (MediaBench)
8	mult_4_4 (UTDSP)	51	trellis (UTDSP)
8	lcdnum (MRTC)	51	fir (MRTC)
9	selection_sort (misc)	52	cjpeg_jpeg6b_transupp (MediaBench)
9	mult_4_4 (UTDSP)	52	trellis (UTDSP)
9	lcdnum (MRTC)	52	fir (MRTC)
9	lmsfir_8_1 (UTDSP)	52	qsort-exam (MRTC)
10	mult_4_4 (UTDSP)	53	trellis (UTDSP)
10	lcdnum (MRTC)	53	fir (MRTC)
10	lmsfir_8_1 (UTDSP)	53	qsort-exam (MRTC)
10	cnt (MRTC)	53	fdct (MRTC)
11	lcdnum (MRTC)	54	fir (MRTC)
11	lmsfir_8_1 (UTDSP)	54	qsort-exam (MRTC)
11	cnt (MRTC)	54	fdct (MRTC)
11	h264dec_ldecode_macroblock (MediaBench)	54	fft_1024 (UTDSP)
12	lmsfir_8_1 (UTDSP)	55	qsort-exam (MRTC)
12	cnt (MRTC)	55	fdct (MRTC)
12	h264dec_ldecode_macroblock (MediaBench)	55	fft_1024 (UTDSP)
12	utdspAdpcm (UTDSP)	55	iir_1_1 (UTDSP)
13	cnt (MRTC)	56	fdct (MRTC)
13	h264dec_ldecode_macroblock (MediaBench)	56	fft_1024 (UTDSP)
13	utdspAdpcm (UTDSP)	56	iir_1_1 (UTDSP)
13	ns (MRTC)	56	v32.modem_achop (UTDSP)
14	h264dec_ldecode_macroblock (MediaBench)	57	fft_1024 (UTDSP)
14	utdspAdpcm (UTDSP)	57	iir_1_1 (UTDSP)
14	ns (MRTC)	57	v32.modem_achop (UTDSP)
14	mult_10_10 (UTDSP)	57	pm (misc)
15	utdspAdpcm (UTDSP)	58	iir_1_1 (UTDSP)
15	ns (MRTC)	58	v32.modem_achop (UTDSP)
15	mult_10_10 (UTDSP)	58	pm (misc)
15	codecs_dcodhuff (misc)	58	jpeg (UTDSP)
16	ns (MRTC)	59	v32.modem_achop (UTDSP)
16	mult_10_10 (UTDSP)	59	pm (misc)
16	codecs_dcodhuff (misc)	59	jpeg (UTDSP)
16	qmf_transmit (UTDSP)	59	g723_encode (misc)
17	mult_10_10 (UTDSP)	60	pm (misc)
17	codecs_dcodhuff (misc)	60	jpeg (UTDSP)
17	qmf_transmit (UTDSP)	60	g723_encode (misc)
17	hamming_window (misc)	60	test3 (misc)
18	codecs_dcodhuff (misc)	61	jpeg (UTDSP)
18	qmf_transmit (UTDSP)	61	g723_encode (misc)
18	hamming_window (misc)	61	test3 (misc)
18	adpcm (MRTC)	61	cover (MRTC)
19	qmf_transmit (UTDSP)	62	g723_encode (misc)
19	hamming_window (misc)	62	test3 (misc)
19	adpcm (MRTC)	62	cover (MRTC)
19	fibcall (MRTC)	62	fir_256_64 (UTDSP)
20	hamming_window (misc)	63	test3 (misc)
20	adpcm (MRTC)	63	cover (MRTC)
20	fibcall (MRTC)	63	fir_256_64 (UTDSP)
20	fir_32_1 (UTDSP)	63	ndes (MRTC)
21	adpcm (MRTC)	64	cover (MRTC)
21	fibcall (MRTC)	64	fir_256_64 (UTDSP)
21	fir_32_1 (UTDSP)	64	ndes (MRTC)
21	v32.modem_eglue (UTDSP)	64	mpeg2 (MediaBench)
22	fibcall (MRTC)	65	fir_256_64 (UTDSP)
22	fir_32_1 (UTDSP)	65	ndes (MRTC)

Set	Benchmark Name	Set	Benchmark Name
22	v32.modem_eglue (UTDSP)	65	mpeg2 (MediaBench)
22	edn (MRTC)	65	g721_encode (misc)
23	fir_32_1 (UTDSP)	66	ndes (MRTC)
23	v32.modem_eglue (UTDSP)	66	mpeg2 (MediaBench)
23	edn (MRTC)	66	g721_encode (misc)
23	h263 (misc)	66	filterbank (StreamIt)
24	v32.modem_eglue (UTDSP)	67	mpeg2 (MediaBench)
24	edn (MRTC)	67	g721_encode (misc)
24	h263 (misc)	67	filterbank (StreamIt)
24	ammunition (misc)	67	g721.marcuslee_decoder (UTDSP)
25	edn (MRTC)	68	g721_encode (misc)
25	h263 (misc)	68	filterbank (StreamIt)
25	ammunition (misc)	68	g721.marcuslee_decoder (UTDSP)
25	utdspCompress (UTDSP)	68	gsm_encode (MediaBench)
26	h263 (misc)	69	filterbank (StreamIt)
26	ammunition (misc)	69	g721.marcuslee_decoder (UTDSP)
26	utdspCompress (UTDSP)	69	gsm_encode (MediaBench)
26	matmult (MRTC)	69	prime (MRTC)
27	ammunition (misc)	70	g721.marcuslee_decoder (UTDSP)
27	utdspCompress (UTDSP)	70	gsm_encode (MediaBench)
27	matmult (MRTC)	70	prime (MRTC)
27	fft_256 (UTDSP)	70	compress (MRTC)
28	utdspCompress (UTDSP)	71	gsm_encode (MediaBench)
28	matmult (MRTC)	71	prime (MRTC)
28	fft_256 (UTDSP)	71	compress (MRTC)
28	jfdctint (MRTC)	71	gsm (MediaBench)
29	matmult (MRTC)	72	prime (MRTC)
29	fft_256 (UTDSP)	72	compress (MRTC)
29	jfdctint (MRTC)	72	gsm (MediaBench)
29	minver (MRTC)	72	sqrt (MRTC)
30	fft_256 (UTDSP)	73	compress (MRTC)
30	jfdctint (MRTC)	73	gsm (MediaBench)
30	minver (MRTC)	73	sqrt (MRTC)
30	lpc (UTDSP)	73	gsm_decode (MediaBench)
31	jfdctint (MRTC)	74	gsm (MediaBench)
31	minver (MRTC)	74	sqrt (MRTC)
31	lpc (UTDSP)	74	gsm_decode (MediaBench)
31	expint (MRTC)	74	spectral (UTDSP)
32	minver (MRTC)	75	sqrt (MRTC)
32	lpc (UTDSP)	75	gsm_decode (MediaBench)
32	expint (MRTC)	75	spectral (UTDSP)
32	fac (MRTC)	75	bsort100 (MRTC)
33	lpc (UTDSP)	76	gsm_decode (MediaBench)
33	expint (MRTC)	76	spectral (UTDSP)
33	fac (MRTC)	76	bsort100 (MRTC)
33	codecs_codrle1 (misc)	76	statemate (MRTC)
34	expint (MRTC)	77	spectral (UTDSP)
34	fac (MRTC)	77	bsort100 (MRTC)
34	codecs_codrle1 (misc)	77	statemate (MRTC)
34	edge_detect (UTDSP)	77	st (MRTC)
35	fac (MRTC)	78	bsort100 (MRTC)
35	codecs_codrle1 (misc)	78	statemate (MRTC)
35	edge_detect (UTDSP)	78	st (MRTC)
35	ludcmp (MRTC)	78	h264dec_ldecode_block (MediaBench)
36	codecs_codrle1 (misc)	79	statemate (MRTC)
36	edge_detect (UTDSP)	79	st (MRTC)
36	ludcmp (MRTC)	79	h264dec_ldecode_block (MediaBench)
36	v32.modem_bencode (UTDSP)	79	ud (MRTC)
37	edge_detect (UTDSP)	80	st (MRTC)
37	ludcmp (MRTC)	80	h264dec_ldecode_block (MediaBench)
37	v32.modem_bencode (UTDSP)	80	ud (MRTC)
37	g721.marcuslee_encoder (UTDSP)	80	qurt (MRTC)
38	ludcmp (MRTC)	81	h264dec_ldecode_block (MediaBench)
38	v32.modem_bencode (UTDSP)	81	ud (MRTC)
38	g721.marcuslee_encoder (UTDSP)	81	qurt (MRTC)
38	latnrm_8_1 (UTDSP)	81	searchmultiarray (misc)

Set	Benchmark Name	Set	Benchmark Name
39	v32.modem_bencode (UTDSP)	82	ud (MRTC)
39	g721.marcuslee_encoder (UTDSP)	82	qurt (MRTC)
39	latnrm_8_1 (UTDSP)	82	searchmultiarray (misc)
39	petrinet (MRTC)	82	lms (MRTC)
40	g721.marcuslee_encoder (UTDSP)	83	qurt (MRTC)
40	latnrm_8_1 (UTDSP)	83	searchmultiarray (misc)
40	petrinet (MRTC)	83	lms (MRTC)
40	janne_complex (MRTC)	83	qmf_receive (UTDSP)
41	latnrm_8_1 (UTDSP)	84	searchmultiarray (misc)
41	petrinet (MRTC)	84	lms (MRTC)
41	janne_complex (MRTC)	84	qmf_receive (UTDSP)
41	latnrm_32_64 (UTDSP)	84	fft1 (MRTC)
42	petrinet (MRTC)	85	lms (MRTC)
42	janne_complex (MRTC)	85	qmf_receive (UTDSP)
42	latnrm_32_64 (UTDSP)	85	fft1 (MRTC)
42	codecs_dcodrle1 (misc)	85	lmsfir_32_64 (UTDSP)
43	janne_complex (MRTC)	86	qmf_receive (UTDSP)
43	latnrm_32_64 (UTDSP)	86	fft1 (MRTC)
43	codecs_dcodrle1 (misc)	86	lmsfir_32_64 (UTDSP)
43	epic (MediaBench)	86	nsichneu (MRTC)

Table E.4. – Benchmark sets for octa-core systems. The order of the benchmarks inside a set denote the assigned core, i.e., the first benchmark is assigned to core 0, the second to core 1, etc.

Set	Benchmark Name	Set	Benchmark Name
1	st (MRTC)	44	cnt (MRTC)
1	filterbank (StreamIt)	44	h264dec_ldecode_macroblock (MediaBench)
1	hamming_window (misc)	44	qsort-exam (MRTC)
1	g723_encode (misc)	44	ndes (MRTC)
1	ud (MRTC)	44	bs (MRTC)
1	codecs_dcodrle1 (misc)	44	v32.modem_achop (UTDSP)
1	ns (MRTC)	44	h263 (misc)
1	lmsfir_8_1 (UTDSP)	44	gsm (MediaBench)
2	filterbank (StreamIt)	45	h264dec_ldecode_macroblock (MediaBench)
2	hamming_window (misc)	45	qsort-exam (MRTC)
2	g723_encode (misc)	45	ndes (MRTC)
2	ud (MRTC)	45	bs (MRTC)
2	codecs_dcodrle1 (misc)	45	v32.modem_achop (UTDSP)
2	ns (MRTC)	45	h263 (misc)
2	lmsfir_8_1 (UTDSP)	45	gsm (MediaBench)
2	qmf_receive (UTDSP)	45	g721.marcuslee_decoder (UTDSP)
3	hamming_window (misc)	46	qsort-exam (MRTC)
3	g723_encode (misc)	46	ndes (MRTC)
3	ud (MRTC)	46	bs (MRTC)
3	codecs_dcodrle1 (misc)	46	v32.modem_achop (UTDSP)
3	ns (MRTC)	46	h263 (misc)
3	lmsfir_8_1 (UTDSP)	46	gsm (MediaBench)
3	qmf_receive (UTDSP)	46	g721.marcuslee_decoder (UTDSP)
3	crc (MRTC)	46	prime (MRTC)
4	g723_encode (misc)	47	ndes (MRTC)
4	ud (MRTC)	47	bs (MRTC)
4	codecs_dcodrle1 (misc)	47	v32.modem_achop (UTDSP)
4	ns (MRTC)	47	h263 (misc)
4	lmsfir_8_1 (UTDSP)	47	gsm (MediaBench)
4	qmf_receive (UTDSP)	47	g721.marcuslee_decoder (UTDSP)
4	crc (MRTC)	47	prime (MRTC)
4	jfdctint (MRTC)	47	bsort100 (MRTC)
5	ud (MRTC)	48	bs (MRTC)
5	codecs_dcodrle1 (misc)	48	v32.modem_achop (UTDSP)
5	ns (MRTC)	48	h263 (misc)
5	lmsfir_8_1 (UTDSP)	48	gsm (MediaBench)
5	qmf_receive (UTDSP)	48	g721.marcuslee_decoder (UTDSP)
5	crc (MRTC)	48	prime (MRTC)

Set	Benchmark Name	Set	Benchmark Name
5	jfdctint (MRTC)	48	bsort100 (MRTC)
5	fir_32_1 (UTDSP)	48	pm (misc)
6	codecs_dcodrle1 (misc)	49	v32.modem_achop (UTDSP)
6	ns (MRTC)	49	h263 (misc)
6	lmsfir_8_1 (UTDSP)	49	gsm (MediaBench)
6	qmf_receive (UTDSP)	49	g721.marcuslee_decoder (UTDSP)
6	crc (MRTC)	49	prime (MRTC)
6	jfdctint (MRTC)	49	bsort100 (MRTC)
6	fir_32_1 (UTDSP)	49	pm (misc)
6	fac (MRTC)	49	fft1 (MRTC)
7	ns (MRTC)	50	h263 (misc)
7	lmsfir_8_1 (UTDSP)	50	gsm (MediaBench)
7	qmf_receive (UTDSP)	50	g721.marcuslee_decoder (UTDSP)
7	crc (MRTC)	50	prime (MRTC)
7	jfdctint (MRTC)	50	bsort100 (MRTC)
7	fir_32_1 (UTDSP)	50	pm (misc)
7	fac (MRTC)	50	fft1 (MRTC)
7	qurt (MRTC)	50	fft_256 (UTDSP)
8	lmsfir_8_1 (UTDSP)	51	gsm (MediaBench)
8	qmf_receive (UTDSP)	51	g721.marcuslee_decoder (UTDSP)
8	crc (MRTC)	51	prime (MRTC)
8	jfdctint (MRTC)	51	bsort100 (MRTC)
8	fir_32_1 (UTDSP)	51	pm (misc)
8	fac (MRTC)	51	fft1 (MRTC)
8	qurt (MRTC)	51	fft_256 (UTDSP)
8	mult_10_10 (UTDSP)	51	edn (MRTC)
9	qmf_receive (UTDSP)	52	g721.marcuslee_decoder (UTDSP)
9	crc (MRTC)	52	prime (MRTC)
9	jfdctint (MRTC)	52	bsort100 (MRTC)
9	fir_32_1 (UTDSP)	52	pm (misc)
9	fac (MRTC)	52	fft1 (MRTC)
9	qurt (MRTC)	52	fft_256 (UTDSP)
9	mult_10_10 (UTDSP)	52	edn (MRTC)
9	fir_256_64 (UTDSP)	52	statemate (MRTC)
10	crc (MRTC)	53	prime (MRTC)
10	jfdctint (MRTC)	53	bsort100 (MRTC)
10	fir_32_1 (UTDSP)	53	pm (misc)
10	fac (MRTC)	53	fft1 (MRTC)
10	qurt (MRTC)	53	fft_256 (UTDSP)
10	mult_10_10 (UTDSP)	53	edn (MRTC)
10	fir_256_64 (UTDSP)	53	statemate (MRTC)
10	iir_1_1 (UTDSP)	53	v32.modem_cnoise (UTDSP)
11	jfdctint (MRTC)	54	bsort100 (MRTC)
11	fir_32_1 (UTDSP)	54	pm (misc)
11	fac (MRTC)	54	fft1 (MRTC)
11	qurt (MRTC)	54	fft_256 (UTDSP)
11	mult_10_10 (UTDSP)	54	edn (MRTC)
11	fir_256_64 (UTDSP)	54	statemate (MRTC)
11	iir_1_1 (UTDSP)	54	v32.modem_cnoise (UTDSP)
11	v32.modem_ddecode (UTDSP)	54	select (MRTC)
12	fir_32_1 (UTDSP)	55	pm (misc)
12	fac (MRTC)	55	fft1 (MRTC)
12	qurt (MRTC)	55	fft_256 (UTDSP)
12	mult_10_10 (UTDSP)	55	edn (MRTC)
12	fir_256_64 (UTDSP)	55	statemate (MRTC)
12	iir_1_1 (UTDSP)	55	v32.modem_cnoise (UTDSP)
12	v32.modem_ddecode (UTDSP)	55	select (MRTC)
12	codecs_codrle1 (misc)	55	g721.marcuslee_encoder (UTDSP)
13	fac (MRTC)	56	fft1 (MRTC)
13	qurt (MRTC)	56	fft_256 (UTDSP)
13	mult_10_10 (UTDSP)	56	edn (MRTC)
13	fir_256_64 (UTDSP)	56	statemate (MRTC)
13	iir_1_1 (UTDSP)	56	v32.modem_cnoise (UTDSP)
13	v32.modem_ddecode (UTDSP)	56	select (MRTC)
13	codecs_codrle1 (misc)	56	g721.marcuslee_encoder (UTDSP)
13	cover (MRTC)	56	matmult (MRTC)

Set	Benchmark Name	Set	Benchmark Name
14	qurt (MRTC)	57	fft_256 (UTDSP)
14	mult_10_10 (UTDSP)	57	edn (MRTC)
14	fir_256_64 (UTDSP)	57	statemate (MRTC)
14	iir_1_1 (UTDSP)	57	v32.modem_cnoise (UTDSP)
14	v32.modem_ddecode (UTDSP)	57	select (MRTC)
14	codecs_codrle1 (misc)	57	g721.marcuslee_encoder (UTDSP)
14	cover (MRTC)	57	matmult (MRTC)
14	fft_1024 (UTDSP)	57	utdspCompress (UTDSP)
15	mult_10_10 (UTDSP)	58	edn (MRTC)
15	fir_256_64 (UTDSP)	58	statemate (MRTC)
15	iir_1_1 (UTDSP)	58	v32.modem_cnoise (UTDSP)
15	v32.modem_ddecode (UTDSP)	58	select (MRTC)
15	codecs_codrle1 (misc)	58	g721.marcuslee_encoder (UTDSP)
15	cover (MRTC)	58	matmult (MRTC)
15	fft_1024 (UTDSP)	58	utdspCompress (UTDSP)
15	trellis (UTDSP)	58	jpeg (UTDSP)
16	fir_256_64 (UTDSP)	59	statemate (MRTC)
16	iir_1_1 (UTDSP)	59	v32.modem_cnoise (UTDSP)
16	v32.modem_ddecode (UTDSP)	59	select (MRTC)
16	codecs_codrle1 (misc)	59	g721.marcuslee_encoder (UTDSP)
16	cover (MRTC)	59	matmult (MRTC)
16	fft_1024 (UTDSP)	59	utdspCompress (UTDSP)
16	trellis (UTDSP)	59	jpeg (UTDSP)
16	sqrt (MRTC)	59	lcdnum (MRTC)
17	iir_1_1 (UTDSP)	60	v32.modem_cnoise (UTDSP)
17	v32.modem_ddecode (UTDSP)	60	select (MRTC)
17	codecs_codrle1 (misc)	60	g721.marcuslee_encoder (UTDSP)
17	cover (MRTC)	60	matmult (MRTC)
17	fft_1024 (UTDSP)	60	utdspCompress (UTDSP)
17	trellis (UTDSP)	60	jpeg (UTDSP)
17	sqrt (MRTC)	60	lcdnum (MRTC)
17	gsm_encode (MediaBench)	60	qmf_transmit (UTDSP)
18	v32.modem_ddecode (UTDSP)	61	select (MRTC)
18	codecs_codrle1 (misc)	61	g721.marcuslee_encoder (UTDSP)
18	cover (MRTC)	61	matmult (MRTC)
18	fft_1024 (UTDSP)	61	utdspCompress (UTDSP)
18	trellis (UTDSP)	61	jpeg (UTDSP)
18	sqrt (MRTC)	61	lcdnum (MRTC)
18	gsm_encode (MediaBench)	61	qmf_transmit (UTDSP)
18	expint (MRTC)	61	mpeg2 (MediaBench)
19	codecs_codrle1 (misc)	62	g721.marcuslee_encoder (UTDSP)
19	cover (MRTC)	62	matmult (MRTC)
19	fft_1024 (UTDSP)	62	utdspCompress (UTDSP)
19	trellis (UTDSP)	62	jpeg (UTDSP)
19	sqrt (MRTC)	62	lcdnum (MRTC)
19	gsm_encode (MediaBench)	62	qmf_transmit (UTDSP)
19	expint (MRTC)	62	mpeg2 (MediaBench)
19	latnrm_8_1 (UTDSP)	62	codecs_dcodhuff (misc)
20	cover (MRTC)	63	matmult (MRTC)
20	fft_1024 (UTDSP)	63	utdspCompress (UTDSP)
20	trellis (UTDSP)	63	jpeg (UTDSP)
20	sqrt (MRTC)	63	lcdnum (MRTC)
20	gsm_encode (MediaBench)	63	qmf_transmit (UTDSP)
20	expint (MRTC)	63	mpeg2 (MediaBench)
20	latnrm_8_1 (UTDSP)	63	codecs_dcodhuff (misc)
20	lms (MRTC)	63	utdspAdpcm (UTDSP)
21	fft_1024 (UTDSP)	64	utdspCompress (UTDSP)
21	trellis (UTDSP)	64	jpeg (UTDSP)
21	sqrt (MRTC)	64	lcdnum (MRTC)
21	gsm_encode (MediaBench)	64	qmf_transmit (UTDSP)
21	expint (MRTC)	64	mpeg2 (MediaBench)
21	latnrm_8_1 (UTDSP)	64	codecs_dcodhuff (misc)
21	lms (MRTC)	64	utdspAdpcm (UTDSP)
21	insertsort (MRTC)	64	cjpeg_jpeg6b_wrbmp (MediaBench)
22	trellis (UTDSP)	65	jpeg (UTDSP)
22	sqrt (MRTC)	65	lcdnum (MRTC)

Set	Benchmark Name	Set	Benchmark Name
22	gsm_encode (MediaBench)	65	qmf_transmit (UTDSP)
22	expint (MRTC)	65	mpeg2 (MediaBench)
22	latnrm_8_1 (UTDSP)	65	codecs_dcodhuff (misc)
22	lms (MRTC)	65	utdspAdpcm (UTDSP)
22	insertsort (MRTC)	65	cjpeg_jpeg6b_wrbmp (MediaBench)
22	petrinet (MRTC)	65	test3 (misc)
23	sqrt (MRTC)	66	lcdnum (MRTC)
23	gsm_encode (MediaBench)	66	qmf_transmit (UTDSP)
23	expint (MRTC)	66	mpeg2 (MediaBench)
23	latnrm_8_1 (UTDSP)	66	codecs_dcodhuff (misc)
23	lms (MRTC)	66	utdspAdpcm (UTDSP)
23	insertsort (MRTC)	66	cjpeg_jpeg6b_wrbmp (MediaBench)
23	petrinet (MRTC)	66	test3 (misc)
23	compress (MRTC)	66	gsm_decode (MediaBench)
24	gsm_encode (MediaBench)	67	qmf_transmit (UTDSP)
24	expint (MRTC)	67	mpeg2 (MediaBench)
24	latnrm_8_1 (UTDSP)	67	codecs_dcodhuff (misc)
24	lms (MRTC)	67	utdspAdpcm (UTDSP)
24	insertsort (MRTC)	67	cjpeg_jpeg6b_wrbmp (MediaBench)
24	petrinet (MRTC)	67	test3 (misc)
24	compress (MRTC)	67	gsm_decode (MediaBench)
24	nsichneu (MRTC)	67	janne_complex (MRTC)
25	expint (MRTC)	68	mpeg2 (MediaBench)
25	latnrm_8_1 (UTDSP)	68	codecs_dcodhuff (misc)
25	lms (MRTC)	68	utdspAdpcm (UTDSP)
25	insertsort (MRTC)	68	cjpeg_jpeg6b_wrbmp (MediaBench)
25	petrinet (MRTC)	68	test3 (misc)
25	compress (MRTC)	68	gsm_decode (MediaBench)
25	nsichneu (MRTC)	68	janne_complex (MRTC)
25	g721_encode (misc)	68	cjpeg_jpeg6b_transupp (MediaBench)
26	latnrm_8_1 (UTDSP)	69	codecs_dcodhuff (misc)
26	lms (MRTC)	69	utdspAdpcm (UTDSP)
26	insertsort (MRTC)	69	cjpeg_jpeg6b_wrbmp (MediaBench)
26	petrinet (MRTC)	69	test3 (misc)
26	compress (MRTC)	69	gsm_decode (MediaBench)
26	nsichneu (MRTC)	69	janne_complex (MRTC)
26	g721_encode (misc)	69	cjpeg_jpeg6b_transupp (MediaBench)
26	h264dec_ldecode_block (MediaBench)	69	minver (MRTC)
27	lms (MRTC)	70	utdspAdpcm (UTDSP)
27	insertsort (MRTC)	70	cjpeg_jpeg6b_wrbmp (MediaBench)
27	petrinet (MRTC)	70	test3 (misc)
27	compress (MRTC)	70	gsm_decode (MediaBench)
27	nsichneu (MRTC)	70	janne_complex (MRTC)
27	g721_encode (misc)	70	cjpeg_jpeg6b_transupp (MediaBench)
27	h264dec_ldecode_block (MediaBench)	70	minver (MRTC)
27	histogram (UTDSP)	70	selection_sort (misc)
28	insertsort (MRTC)	71	cjpeg_jpeg6b_wrbmp (MediaBench)
28	petrinet (MRTC)	71	test3 (misc)
28	compress (MRTC)	71	gsm_decode (MediaBench)
28	nsichneu (MRTC)	71	janne_complex (MRTC)
28	g721_encode (misc)	71	cjpeg_jpeg6b_transupp (MediaBench)
28	h264dec_ldecode_block (MediaBench)	71	minver (MRTC)
28	histogram (UTDSP)	71	selection_sort (misc)
28	fdct (MRTC)	71	v32.modem_eglue (UTDSP)
29	petrinet (MRTC)	72	test3 (misc)
29	compress (MRTC)	72	gsm_decode (MediaBench)
29	nsichneu (MRTC)	72	janne_complex (MRTC)
29	g721_encode (misc)	72	cjpeg_jpeg6b_transupp (MediaBench)
29	h264dec_ldecode_block (MediaBench)	72	minver (MRTC)
29	histogram (UTDSP)	72	selection_sort (misc)
29	fdct (MRTC)	72	v32.modem_eglue (UTDSP)
29	edge_detect (UTDSP)	72	fir (MRTC)
30	compress (MRTC)	73	gsm_decode (MediaBench)
30	nsichneu (MRTC)	73	janne_complex (MRTC)
30	g721_encode (misc)	73	cjpeg_jpeg6b_transupp (MediaBench)
30	h264dec_ldecode_block (MediaBench)	73	minver (MRTC)

Set	Benchmark Name	Set	Benchmark Name
30	histogram (UTDSP)	73	selection_sort (misc)
30	fdct (MRTC)	73	v32.modem_eglue (UTDSP)
30	edge_detect (UTDSP)	73	fir (MRTC)
30	ammunition (misc)	73	spectral (UTDSP)
31	nsichneu (MRTC)	74	janne_complex (MRTC)
31	g721_encode (misc)	74	cjpeg_jpeg6b_transupp (MediaBench)
31	h264dec_ldecode_block (MediaBench)	74	minver (MRTC)
31	histogram (UTDSP)	74	selection_sort (misc)
31	fdct (MRTC)	74	v32.modem_eglue (UTDSP)
31	edge_detect (UTDSP)	74	fir (MRTC)
31	ammunition (misc)	74	spectral (UTDSP)
31	fibcall (MRTC)	74	mult_4_4 (UTDSP)
32	g721_encode (misc)	75	cjpeg_jpeg6b_transupp (MediaBench)
32	h264dec_ldecode_block (MediaBench)	75	minver (MRTC)
32	histogram (UTDSP)	75	selection_sort (misc)
32	fdct (MRTC)	75	v32.modem_eglue (UTDSP)
32	edge_detect (UTDSP)	75	fir (MRTC)
32	ammunition (misc)	75	spectral (UTDSP)
32	fibcall (MRTC)	75	mult_4_4 (UTDSP)
32	adpcm (MRTC)	75	lpc (UTDSP)
33	h264dec_ldecode_block (MediaBench)	76	minver (MRTC)
33	histogram (UTDSP)	76	selection_sort (misc)
33	fdct (MRTC)	76	v32.modem_eglue (UTDSP)
33	edge_detect (UTDSP)	76	fir (MRTC)
33	ammunition (misc)	76	spectral (UTDSP)
33	fibcall (MRTC)	76	mult_4_4 (UTDSP)
33	adpcm (MRTC)	76	lpc (UTDSP)
33	latnrm_32_64 (UTDSP)	76	lmsfir_32_64 (UTDSP)
34	histogram (UTDSP)	77	selection_sort (misc)
34	fdct (MRTC)	77	v32.modem_eglue (UTDSP)
34	edge_detect (UTDSP)	77	fir (MRTC)
34	ammunition (misc)	77	spectral (UTDSP)
34	fibcall (MRTC)	77	mult_4_4 (UTDSP)
34	adpcm (MRTC)	77	lpc (UTDSP)
34	latnrm_32_64 (UTDSP)	77	lmsfir_32_64 (UTDSP)
34	audiobeam (StreamIt)	77	epic (MediaBench)
35	fdct (MRTC)	78	v32.modem_eglue (UTDSP)
35	edge_detect (UTDSP)	78	fir (MRTC)
35	ammunition (misc)	78	spectral (UTDSP)
35	fibcall (MRTC)	78	mult_4_4 (UTDSP)
35	adpcm (MRTC)	78	lpc (UTDSP)
35	latnrm_32_64 (UTDSP)	78	lmsfir_32_64 (UTDSP)
35	audiobeam (StreamIt)	78	epic (MediaBench)
35	iir_4_64 (UTDSP)	78	ludcmp (MRTC)
36	edge_detect (UTDSP)	79	fir (MRTC)
36	ammunition (misc)	79	spectral (UTDSP)
36	fibcall (MRTC)	79	mult_4_4 (UTDSP)
36	adpcm (MRTC)	79	lpc (UTDSP)
36	latnrm_32_64 (UTDSP)	79	lmsfir_32_64 (UTDSP)
36	audiobeam (StreamIt)	79	epic (MediaBench)
36	iir_4_64 (UTDSP)	79	ludcmp (MRTC)
36	v32.modem_bencode (UTDSP)	79	searchmultiarray (misc)
37	ammunition (misc)	80	spectral (UTDSP)
37	fibcall (MRTC)	80	mult_4_4 (UTDSP)
37	adpcm (MRTC)	80	lpc (UTDSP)
37	latnrm_32_64 (UTDSP)	80	lmsfir_32_64 (UTDSP)
37	audiobeam (StreamIt)	80	epic (MediaBench)
37	iir_4_64 (UTDSP)	80	ludcmp (MRTC)
37	v32.modem_bencode (UTDSP)	80	searchmultiarray (misc)
37	cnt (MRTC)	80	st (MRTC)
38	fibcall (MRTC)	81	mult_4_4 (UTDSP)
38	adpcm (MRTC)	81	lpc (UTDSP)
38	latnrm_32_64 (UTDSP)	81	lmsfir_32_64 (UTDSP)
38	audiobeam (StreamIt)	81	epic (MediaBench)
38	iir_4_64 (UTDSP)	81	ludcmp (MRTC)
38	v32.modem_bencode (UTDSP)	81	searchmultiarray (misc)

Set	Benchmark Name	Set	Benchmark Name
38	cnt (MRTC)	81	st (MRTC)
38	h264dec_ldecode_macroblock (MediaBench)	81	filterbank (StreamIt)
39	adpcm (MRTC)	82	lpc (UTDSP)
39	latnrm_32_64 (UTDSP)	82	lmsfir_32_64 (UTDSP)
39	audiobeam (StreamIt)	82	epic (MediaBench)
39	iir_4_64 (UTDSP)	82	ludcmp (MRTC)
39	v32.modem_bencode (UTDSP)	82	searchmultiarray (misc)
39	cnt (MRTC)	82	st (MRTC)
39	h264dec_ldecode_macroblock (MediaBench)	82	filterbank (StreamIt)
39	qsort-exam (MRTC)	82	hamming_window (misc)
40	latnrm_32_64 (UTDSP)	83	lmsfir_32_64 (UTDSP)
40	audiobeam (StreamIt)	83	epic (MediaBench)
40	iir_4_64 (UTDSP)	83	ludcmp (MRTC)
40	v32.modem_bencode (UTDSP)	83	searchmultiarray (misc)
40	cnt (MRTC)	83	st (MRTC)
40	h264dec_ldecode_macroblock (MediaBench)	83	filterbank (StreamIt)
40	qsort-exam (MRTC)	83	hamming_window (misc)
40	ndes (MRTC)	83	g723_encode (misc)
41	audiobeam (StreamIt)	84	epic (MediaBench)
41	iir_4_64 (UTDSP)	84	ludcmp (MRTC)
41	v32.modem_bencode (UTDSP)	84	searchmultiarray (misc)
41	cnt (MRTC)	84	st (MRTC)
41	h264dec_ldecode_macroblock (MediaBench)	84	filterbank (StreamIt)
41	qsort-exam (MRTC)	84	hamming_window (misc)
41	ndes (MRTC)	84	g723_encode (misc)
41	bs (MRTC)	84	ud (MRTC)
42	iir_4_64 (UTDSP)	85	ludcmp (MRTC)
42	v32.modem_bencode (UTDSP)	85	searchmultiarray (misc)
42	cnt (MRTC)	85	st (MRTC)
42	h264dec_ldecode_macroblock (MediaBench)	85	filterbank (StreamIt)
42	qsort-exam (MRTC)	85	hamming_window (misc)
42	ndes (MRTC)	85	g723_encode (misc)
42	bs (MRTC)	85	ud (MRTC)
42	v32.modem_achop (UTDSP)	85	codecs_dcodrle1 (misc)
43	v32.modem_bencode (UTDSP)	86	searchmultiarray (misc)
43	cnt (MRTC)	86	st (MRTC)
43	h264dec_ldecode_macroblock (MediaBench)	86	filterbank (StreamIt)
43	qsort-exam (MRTC)	86	hamming_window (misc)
43	ndes (MRTC)	86	g723_encode (misc)
43	bs (MRTC)	86	ud (MRTC)
43	v32.modem_achop (UTDSP)	86	codecs_dcodrle1 (misc)
43	h263 (misc)	86	ns (MRTC)

TDMA Slot Length Evaluation

F

The low-level static instruction memory allocation for TDMA-based multi-core systems presented in Chapter 4 requires the TDMA slots of all cores to be equally sized to the minimum slot length of a single net access latency of the shared memory. This rather strict appearing requirement in order to minimize the uncertainty of a shared memory access raises the question of its usefulness in practice. The correlation of average-case execution times and maximum number of shared memory accesses inside a TDMA-scheduled architecture is therefore examined. The results of this case-study are depicted in Figure F.1. These results were first published in the second issue of the 19th volume of the *ACM Transactions on Embedded Computing Systems* in 2020 [LOF20].

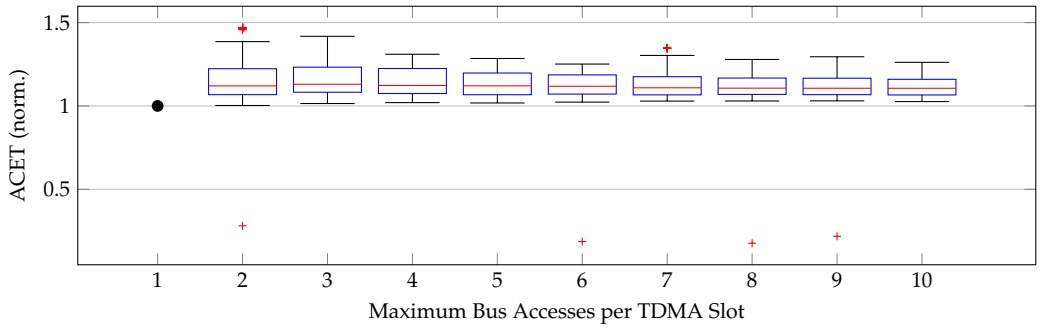


Figure F.1. – Normalized average-case execution times for varying numbers of accesses per TDMA slot. ACETs were generated with the cycle-true instruction set simulator CoMET [Syn18] applied to the ARM7TDMI multi-core model described in Section 4.2.1.

The graph shows the distribution of average-case execution times with respect to the maximum number of bus accesses per TDMA slot, normalized to the timing when using minimum TDMA slot lengths. The central mark of each box denotes the median, while the edges depict the 25th and 75th percentiles. The maximum whisker length is defined as 1.5 times the difference between the 75th and 25th percentile. For each TDMA slot setting, 153 dualcore, 151 quadcore and 147 octacore systems were evaluated with benchmarks bundled to multi-core packages from the MRTC- [GBEL10], UTDSP- [LCS92], DSPStone- [ZVSM94], MediaBench- [LPM97], MiBench- [GRE+01], NetBench- [MMSH01], PolyBench- [Lou] and StreamIT-benchmark suite [Str18]. The single-core benchmarks were bundled to multi-core packages (one task per core) such that all benchmarks used in a bundle have a similar single-core ACET. The ACET of a given multi-core system was analyzed by CoMET [Syn18] and an existing ARM7TDMI multi-core model [Kel15] which resembles the system described in Section 4.2.1. No specific SPM allocation was done, hence all instructions reside in the shared Flash memory.

Example: A benchmark took on average (median) 1.12 times longer to execute when setting all TDMA slots of a multi-core system to the length of 2 Flash memory accesses (cf. 2 on the x-axis) than with a setting of only 1 Flash memory access per slot. Figure F.1 shows that beside 4 outliers, none of the evaluated benchmarks show a degraded ACET when using a minimum TDMA slot length (1 shared memory access per slot). Accordingly, we assume that restricting all TDMA slots to a minimum slot length of a single shared memory access latency does typically not degrade the timing of a system and is therefore a valid constraint for our optimization.

G

Evolutionary Algorithm-based Memory Allocation Mutation Probability Evaluation

To evaluate the influence of the mutation probability on the quality of the results of the EA-based static memory allocation, a variety of different mutation probabilities are tested. For this, the EA-based static instruction memory allocation described in Appendix C is exemplary applied on the benchmarks of the MRTC suite [GBEL10] for a single-core ARM7TDMI-based architecture with the instruction SPM set to 50% relative to the benchmark size. Mutation probabilities from 0.01 to 0.1 are evaluated in 0.01 steps. The evolutionary algorithm is set to 20 generations with 20 individuals per generation. Each test is carried out 5 times to mitigate statistical spikes.

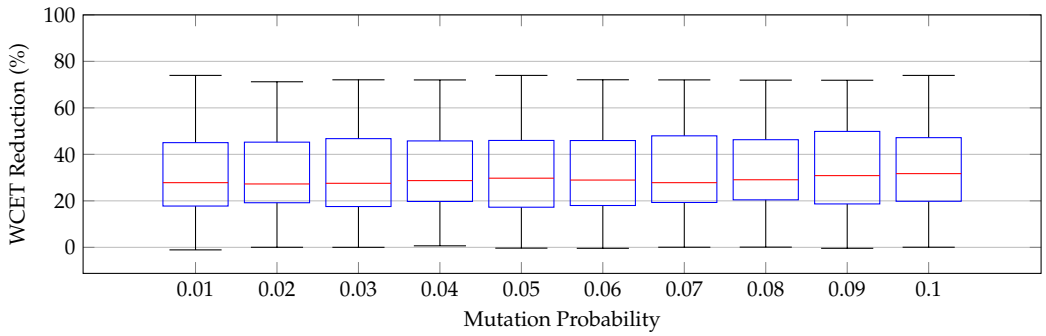


Figure G.1. – WCET reduction in percent after applying the EA-based static instruction memory allocation depending on the mutation probability.

The results of this evaluation is shown using the box plots in Figure G.1. The central mark of each box denotes the median, while the edges depict the 25th and 75th percentiles. The maximum whisker length is defined as 1.5 times the difference between the 75th and 25th percentile. Although there are some very slight differences, no clear correlation between the quality of the results and the mutation probability can be seen here.

Figure G.2 shows the average runtimes of the EA-based static instruction memory allocation depending on the mutation probability. The box plot presentation of the previous graph is not chosen here, as the runtimes between the different benchmarks vary greatly (from just over a minute up to an hour). It is noticeable that the evolutionary algorithms finish the fastest on average for a mutation probability of 0.01 and second fastest for 0.02.

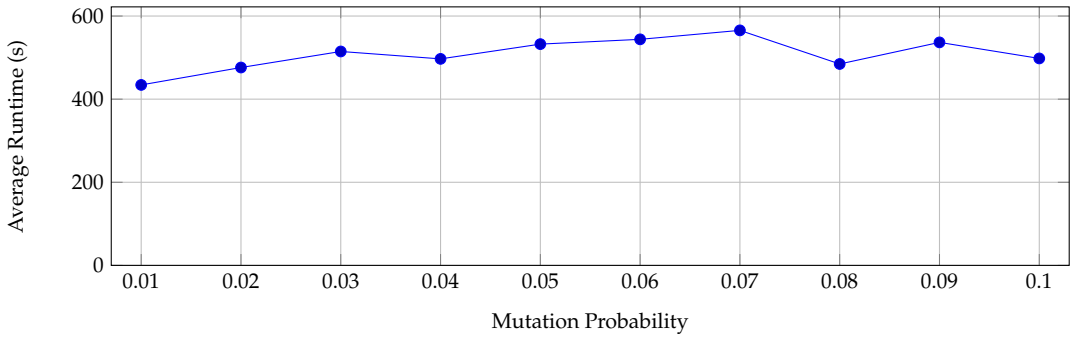


Figure G.2. – Average runtimes of the EA-based static instruction memory allocation depending on the mutation probability.

H

WCET Improvement of Sub Basic Block Splitting

Figure H.1 shows the average improvement in WCET when applying sub basic block splitting before performing the ILP-based bus-unaware instruction SPM allocation, depending on the relative instruction SPM size. Here, over 150 benchmarks were evaluated for a single-core single-task system with SPM sizes varying from 10% to 90% relative to each benchmark's size. For each benchmark and SPM size combination, the ILP-based bus-unaware instruction SPM allocation was performed two separate times: Once with sub basic block splitting applied and once without. The sub basic block splitting cuts each data accessing instruction into its own sub basic block, potentially increasing the number of basic blocks of a benchmark and refining the granularity for the instruction allocation. An ARM7TDMI was used here as an evaluation platform. The y-axis shows the average WCET improvement due to the sub basic block splitting. As an example, the WCET of a benchmark was on average 1.76% lower after performing the instruction allocation *with* sub basic block splitting applied than without at a relative SPM size of 20%.

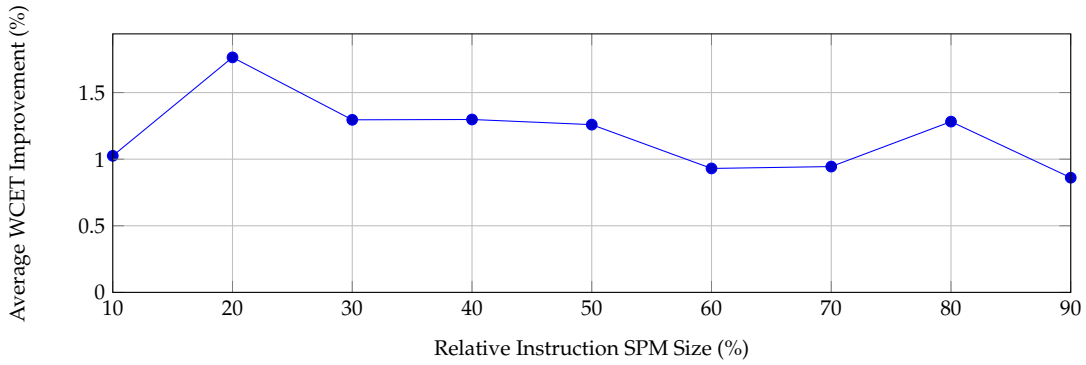


Figure H.1. – Average WCET improvement due to sub basic block splitting for the ILP-based bus-unaware instruction SPM allocation depending on the relative instruction SPM size for single-core single-task systems.