Bus-Aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems*

Dominic Oehlert[1], Arno Luppold[2], and Heiko Falk[3]

1 **Hamburg University of Technology, Hamburg, Germany**
`dominic.oehlert@tuhh.de`
2 **Hamburg University of Technology, Hamburg, Germany**
`arno.luppold@tuhh.de`
3 **Hamburg University of Technology, Hamburg, Germany**
`heiko.falk@tuhh.de`

Abstract

Over the past years, multicore systems emerged into the domain of hard real-time systems. These systems introduce common buses and shared memories which heavily influence the timing behavior. We show that existing WCET optimizations may lead to suboptimal results when applied to multicore setups. Additionally we provide both a genetic and a precise Integer Linear Programming (ILP)-based static instruction scratchpad memory allocation optimization which are capable of exploiting multicore properties, resulting in a WCET reduction of 26% in average compared with a bus-unaware optimization. Furthermore, we show that our ILP-based optimization's average runtime is distinctively lower in comparison to the genetic approach. Although limiting the number of tasks per core to one and partially exploiting private instruction SPMs, we cover the most crucial elements of a multicore setup: the interconnection and shared resources.

1998 ACM Subject Classification C.3 Special-Purpose and Application-Based Systems, D.3.4 Processors, G.1.6 Optimization

Keywords and phrases compiler, optimization, WCET, real-time, multicore

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.1

1 Introduction

The Worst-Case Execution Time (WCET) of a program is defined as the worst possible time the program needs from the start until the end of its execution. In hard real-time systems where a task must provably finish its execution within a given amount of time, reducing the WCET is crucial to the correct behavior of the system.

Within the last couple of years, increasing computational requirements led to the introduction of multicore systems into the world of hard real-time systems. The drawback of these systems is the much more complex timing analysis due to shared memories which are accessed over common bus systems. Due to these common buses, the WCET of a program is heavily influenced by the implemented bus scheduling policy. These effects have to be taken into account during the analysis in order to determine a safe, yet tight WCET. Recent scientific works [3, 11] tackle the precise analysis of these systems, but optimization techniques have yet to catch up to these new challenges.

* This work received funding from Deutsche Forschungsgemeinschaft (DFG) under grant FA 1017/1-2. This work was partially supported by COST Action IC1202: Timing Analysis On Code-Level (TACLe).

29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna; Article No. 1; pp. 1:1–1:22

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Over the last years, memory optimizations, especially those featuring a fast but small Scratchpad Memory (SPM) have proven to be powerful tools to selectively optimize the WCET of a program for singlecore systems [5, 18]. We therefore exemplary use a state of the art ILP-based static WCET-aware instruction SPM allocation for singlecore systems to show that those optimizations may yield suboptimal results when applied to multicore systems without taking common memory buses into account. This even holds for relatively simple bus access algorithms like TDMA with equally-sized fixed slot lengths.

To counter these issues, we extend the ILP model by a precise bus model for a TDMA schedule with fixed slot lengths and show to be able to specifically optimize programs for multicore systems. Our experiments show WCET reductions of up to 90% percent over the bus-unaware SPM allocation. The presented approach covers precisely the potential blocking times of an access to a shared memory due to the TDMA schedule. As reference for the assessment of the quality of the ILP-based model, we additionally propose a bus-aware SPM allocation based on a genetic algorithm.

This paper is outlined as follows: Section 2 gives an overview over the related work. In Section 3, we give an overview of the used multicore architecture. Section 4 introduces a motivating example, illustrating the necessity to consider bus-related effects during WCET-driven optimizations for a multicore platform. In Section 5, we present the used base ILP model, the bus-aware extensions, as well as an overview of the nomenclature and certain preliminaries. Section 6 presents the evolutionary algorithm used for an instruction SPM allocation on a multicore platform. The bus-aware extensions to the ILP model and the evolutionary-based approach form the contributions of this paper. The evaluation of the presented approaches is shown in Section 7. Section 8 concludes this paper and gives an outlook on possible future work.

2 Related Work

Regarding hard real-time multicore systems, numerous parameters can be taken into account in order to increase its performance. Kelter et al. [10] presented WCET-aware scheduling optimizations for multicore systems. They showed a WCET-aware optimization centered on the schedule parameters of a system, e.g., the bus scheduling policy, number of bus slots or priorities of slots. Besides, they also presented an optimization featuring a WCET-oriented instruction reordering. For both approaches, evolutionary algorithms were exploited.

Suhendra and Mitra [17] presented the effects of locking and partitioning caches inside a multicore architecture, regarding the worst-case performance. They examined the timing profits of locking or partitioning a shared L2 instruction cache, based on task or core level. Both, dynamic and static cache locking, were discussed.

The optimization of programs in hard real-time systems using scratchpad memory allocation has been discussed in several publications. An ILP-based optimization for a WCET-aware data scratchpad memory (SPM) allocation for a singlecore architecture was presented by Suhendra et al. [18]. Based upon this structure, an adapted version for instruction memory allocation was introduced by Falk and Kleinsorge [5].

Liu and Zhang [15] presented different multicore architectures featuring multilevel scratchpad memories. They also demonstrated an ILP-based optimization to decrease the WCET of a program by allocating certain parts of a program to the different SPMs available. Static and dynamic SPM allocations were discussed, while also an evaluation concerning the worst-case energy consumption was given. However, bus- or multicore-related factors like

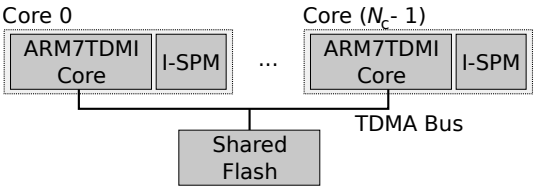

Figure 1 Overview of the proposed multicore architecture.

bus communication latencies are neglected in the presented timing models, thus lowering the accuracy of the overall model.

Kim et al. [13] presented a WCET-aware approach for dynamic code management on SPMs that focused on software-managed multicores. They proposed a multicore architecture with private SPMs, in which every main memory access is forced to go through the SPM, which issues a direct memory access (DMA). Based on this system, an ILP-based and a heuristic technique to reduce the WCET of a program were presented. Also here, the interconnection network between the SPMs and the main memory is neglected in terms of timing, thus degrading the accuracy of the presented model.

Kafshdooz and Ejlali [9] presented an ILP- and heuristic-based approach in order to reduce the WCET of a program, exploiting dynamic SPM allocation in a multicore-multiprocess system. Each bus access was assumed to take the worst latency possible, leading to a heavy over-approximation.

Chattopadhyay and Roychoudhury [2] introduced a static scratchpad allocation on a multicore system, that features bus-awareness. They presented a heuristic approach to reduce the worst-case response time (WCRT) of a multiprocessor program, based on a bus-aware WCRT-analysis. This analysis result was iteratively used to find a proper SPM allocation.

In contrast to the discussed approaches, we present an ILP-model which features a precise bus-awareness without the requirement to rely on the worst-case timing for each access. Besides, we demonstrate an evolutionary-based approach in order to classify the figure of merit of the presented model.

3 Multicore Architecture

This section presents the architecture used throughout this paper which is illustrated in Figure 1. The system consists of N_c parallel homogeneous cores with private instruction SPMs, a TDMA scheduled bus and a shared Flash memory which is connected to the bus. An SPM typically consists of a static RAM which is placed closely to the processor, leading to significantly lower access time in comparison to, e.g., Flash memories, yet limiting their overall capacity. We assume one task per core. Due to the homogeneity of the cores and the TDMA scheduled bus, the mapping of tasks to a core is not covered in this paper. The ARM7TDMI core was used for evaluation purposes only. The presented ILP-model is, however, generally applicable to other multicore architectures based on in-order processors. To improve the predictability, all caches of the cores are disabled. The instruction SPM of each core is private and can only be accessed via the attached core. Hence, no bus access is necessary during a scratchpad memory access. The shared Flash memory has to be accessed via the bus. The access delay of the Flash memory (excluding possible stalls before a bus grant) is considerably higher (approx. factor 6) in comparison to the access delay of an SPM. The bus is assumed to be TDMA scheduled, while the TDMA schedule consists of N_c slots. Each core's time slot length, during which it exclusively can access the bus, can be adjusted

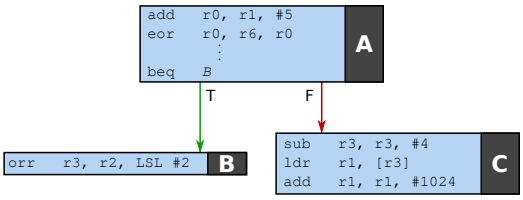

Figure 2 Exemplary CFG I.

Table 1 WCET (in cycles) for the exemplary program.

Basic Block	Flash	SPM
A	390	20
B	96	1
C	114	9

individually. It is assumed that all cores are globally synchronous. 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.

The ARM7TDMI architecture features a 3 staged pipeline, fetches each instruction piecewise and supports a very basic form of branch „prediction“ (always not taken, even if unconditional).

4 Motivating Example

In order to demonstrate the necessity to consider the bus-related effects while performing an SPM allocation, we are assuming a small exemplary program, represented by the control flow graph (CFG) shown in Figure 2.

Basic block (BB) A is not shown fully here due to space limitations and consists of 20 instructions in total. The second instruction (`ldr`) of basic block C 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 3 with 4 parallel cores and a TDMA bus with equally-sized slot lengths (each slot can accommodate 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 20 Bytes. We assume the program to be executed on core 0, which owns the first bus slot of every bus period inside the schedule.

Following the ILP-based static instruction SPM allocation for a singlecore system [5], the WCET of the program is analyzed twice, once with all BBs placed inside the shared Flash memory and once with all BBs placed inside the (virtually enlarged) private SPM of core 0. These analyses return the WCET per BB denoted in Table 1. The BBs allocated to the SPM are loaded into the SPM once prior to the actual execution of the program. Therefore, the time required to initially load the BBs into the SPM can be neglected.

Since only basic blocks A and C are part of the worst case execution path (WCEP), but the SPM has a limitation of only 20 Bytes (5 instructions), the ILP solver will decide to place BB C into the SPM, resulting in an expected WCET reduction of 84 cycles. This reduction already includes jump correction costs of 21 cycles. These additional cycles stem from jump correction code, which has to be inserted subsequent to the SPM allocation decisions in order to restore a working control flow. Since BB C is placed in a different memory region, it is not subsequent (by means of physical addresses) to BB A anymore. Hence, an explicit jump (most likely an indirect one to cope with the physical address offsets of the memory regions) to C is required to be inserted at the end of BB A. However, a final WCET analysis with the proposed SPM allocation done reveals a WCET of 511 cycles, resulting in an actually worse timing after the optimization than before (504 cycles).

The cause of this misprediction is based on the bus-related latencies. Figure 3a shows the WCET of the program in regard to the bus schedule in case BB A and C are allocated to the shared Flash memory. The individual bus slots are shown along the x-axis, with one bus

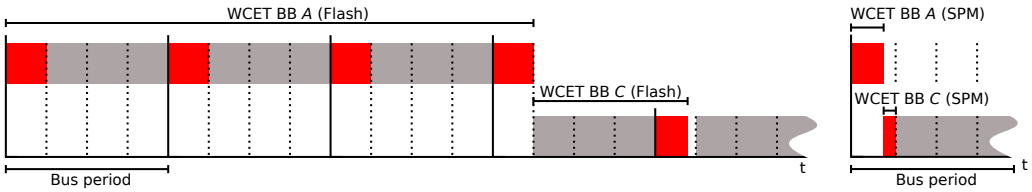

Figure 3 WCETs of BB A and C in regard to the bus schedule: (a) allocated to Flash, (b) allocated to SPM.

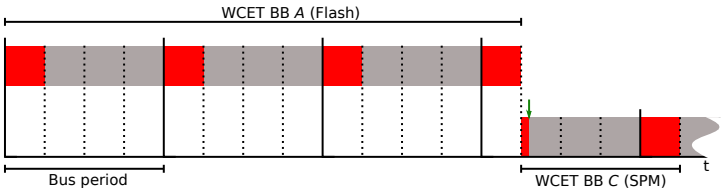

Figure 4 WCETs of BB A and C in regard to the bus schedule, „optimized“ Allocation.

period consisting of 4 bus slots. Core 0 (on which the program is allocated) owns the first bus slot of every period. The areas marked in red show the actual execution of instructions. Basic block A can be executed in exactly four slots, while BB C starts at the beginning of the second slot of a bus period, hence the processor has to stall until another fetch can be done. When the bus grant is regained, BB C can be executed within one bus slot.

Figure 3b shows the WCET for the second step, when all BBs have been allocated into the private SPM. Due to the severely lower access times to the SPM, basic blocks A and B can now be completely executed in the first bus slot.

Based upon these timings, the ILP solver assigns BB C to the SPM. The actual WCET analysis of the allocated program in regard to the bus schedule can be seen in Figure 4. Basic block A is executed in the same manner as seen in Figure 3a.

However, in difference to Figure 3b, the execution of basic block C now starts at the beginning of the second bus slot. This is caused by the fact that the execution time of the preceding basic block A is different to the timing analyzed when the whole program was placed inside the SPM. The execution of BB C can be started, since the instructions are placed inside the private SPM, but has to be stalled during the second instruction (`ldr`, depicted with an arrow) until the `.data` section can be accessed. Additionally, jump correction code has to be inserted to create a memory region crossing jump from basic block A to C. These circumstances lead to a drastically higher WCET for basic block C than expected, which again leads to a higher WCET in total.

This example shows the crucial sensitivity of a program's timing in regard to the underlying interconnection network. Neglecting bus-related timings strongly decreases the accuracy of the SPM allocation optimization, thus easily resulting in an underestimated WCET. We see that even though the presented optimization methods work fine for singlecore platforms, it does not give an accurate result when being applied to a multicore platform. This is due to the fact that the ILP model does not have a required notion of history to consider timing effects which are induced by preceding allocation decisions.

5 ILP Model

In the following, we will present an ILP model which is able to predict the timing behavior of the used bus architecture, thus enabling a WCET-centered instruction SPM allocation on multicore platforms, avoiding mispredictions as shown in the previous section. First, we will give a short overview of the notational conventions and ILP formulations used throughout this paper. Subsequently, we will shortly introduce the base ILP model which our work builds upon. Eventually, we are presenting our bus-aware extensions, which are enabling a precise prediction of bus-related timing behaviors.

5.1 Notational Conventions

In the following, lower case italic Latin letters like i will be used for ILP variables. Upper case italic Latin letters like A represent constants inside the ILP model. Letters in bold, e.g., $\boldsymbol{o}$, depict intervals of bus offsets. Lastly, lower case italic Greek letters like ν are used to denote a certain basic block.

Table 2 contains all ILP variables used in this paper, while Table 3 contains other miscellaneous symbols and their description used in this paper.

5.2 Mathematical Preliminaries and ILP Formulations

In this section, we will present certain mathematical preliminaries and ILP formulations in a general way which are used throughout the paper.

Modulo Function

We are using the definition of a modulo function described by Knuth [14]:

$$m = x \bmod y = x - \left\lfloor \frac{x}{y} \right\rfloor \cdot y \text{ , if } y \neq 0 \tag{1}$$

where x and y are any integer numbers. The resulting value m has the same sign as the divisor y. We reformulate (1) to the following equations:

$$x < (q+1) \cdot y \tag{2}$$
$$x \geq y \cdot q \tag{3}$$
$$m = x - q \cdot y \tag{4}$$

Equations (2) and (3) implement the floored division with q holding the result of $\lfloor \frac{x}{y} \rfloor$. Variable y is assumed to be always non-zero. The variable m is set to the modulo result by Equation (4). In case we assume the divisor y to be constant, equations (2) - (4) are linear and can be used inside an ILP formulation.

Conditional Assignment of ILP Variables

The following conditional assignment is given:

$$a = \begin{cases} u & \text{if } c = 1, \\ v & \text{else.} \end{cases} \tag{5}$$

Table 2 ILP decision variables used

Symbol	Description
d_ν	Additional number of stall cycles preceding an explicit data access during the execution of block ν in comparison to the analysis results.
$l_{\nu,\mu}$	Additional number of cycles needed due to bus stalling during the execution of jump correction code from block ν to μ in comparison to the analyzed interval. This variable does not contain the cycles required for the execution of the code, solely the cycles needed to gain the first grant.
$\boldsymbol{o}_\nu^{\text{In}}$	The incoming bus offset interval at ν.
$\boldsymbol{o}_{\nu,\mu,\text{W}}^{\text{Out}}$	Interval variable representing the outgoing bus offset interval at BB ν *with* considering potential jump correction to its successor BB μ.
$\boldsymbol{o}_{\nu,\text{WO}}^{\text{Out}}$	Interval variable representing the outgoing bus offset interval at BB ν *without* considering a potential jump correction.
$o_{\nu,\text{low}}$, $o_{\nu,\text{high}}$	The lower and upper elements of the corresponding bus offset interval $\boldsymbol{o}_\nu$.
r_ν^{Data}	The number of cycles required to receive a bus grant to access shared data at the BB ν.
r_ν^{Jump}	The number of cycles required to receive a bus grant block during the execution of jump correction code at BB ν.
w_ν	The accumulated WCET starting at BB ν.
x_ν	Binary variable representing whether BB ν is assigned to the SPM (x_ν=1) or not.

Table 3 Miscellaneous symbols used

Symbol	Description
$\boldsymbol{A}_{\nu,\text{Flash}}^{\text{In}}$, ($\boldsymbol{A}_{\nu,\text{SPM}}^{\text{In}}$)	Incoming bus offset interval at BB ν if the whole program is allocated to Flash (SPM).
$\boldsymbol{A}_{\nu,\text{Flash}}^{\text{Out}}$, ($\boldsymbol{A}_{\nu,\text{SPM}}^{\text{Out}}$)	Outgoing bus offset interval at BB ν if the whole program is allocated to Flash (SPM).
$C_{\nu,\text{Flash}}$,($C_{\nu,\text{SPM}}$)	WCET of 1 execution of BB ν when allocated to the Flash memory (SPM).
F_{SPM}	Access delay of the SPM.
F_{Flash}	Access delay of the Flash memory (bus grant acquired).
G_ν	Expected timing gain if a BB ν is assigned to the SPM in comparison to a Flash allocation.
H_ν	Binary constant set to 1 in case BB ν contains an instruction which potentially accesses the `.data` section
I	A core ID (0 ... N_c-1).
$J_{\mu,\nu,\text{SPM}}$	Cycles needed for the execution of jump correction code from BB ν (in SPM) to μ (in Flash), neglecting the required pipeline refill.
N_c	Number of cores.
P	Total bus period in cycles.
$Q_{\text{Flash}\rightarrow\text{SPM}}$	Bus offset after the execution of jump correction code in case the source BB is placed inside the Flash memory and its target BB resides in the SPM.
R_ν^{Data}	Required number of stall cycles to receive the bus grant for the shared memory access at block ν, accounted by the analysis.
S_ν	Code size of basic block ν.
S_{SPM}	Total size of a private SPM.
$\boldsymbol{T}_{\nu,\text{SPM}}$	Execution time window of BB ν as a result of an BCET/WCET analysis when the whole program is allocated to the SPM.
ν, μ	Indexes representing BBs.

with a, u, v and c being ILP variables, whereas the condition variable c is restricted to Boolean values. We are expressing equation (5) as a set of inequations in order to formulate if-then-else structures inside an ILP model.

$$a \geq u - (1 - c) \cdot M \tag{6}$$
$$a \leq u + (1 - c) \cdot M \tag{7}$$
$$a \geq v - c \cdot M \tag{8}$$
$$a \leq v + c \cdot M \tag{9}$$

We are using the so-called big-M method, where M is a sufficiently large constant. Equations (6) and (7) 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 (8) and (9) force the 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:

$$\max(x, y) \tag{10}$$
$$\min(x, y) \tag{11}$$

with x and y being ILP variables. In order to express these two functions in terms of ILP constraints, we first create an ILP variable c restricted to Boolean values, used as a condition.

$$y \leq x + c \cdot M \tag{12}$$
$$x \leq y + (1 - c) \cdot M \tag{13}$$

M is a sufficiently large constant. Equations (12) and (13) 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:

$$\max(x, y) = \begin{cases} y & \text{if } c = 1, \\ x & \text{else.} \end{cases} \tag{14}$$

$$\min(x, y) = \begin{cases} x & \text{if } c = 1, \\ y & \text{else.} \end{cases} \tag{15}$$

The case statement used in equations (14) and (15) can be represented using the conditional assignment of ILP variables as shown before.

5.3 Base Model

The optimization introduced by Falk and Kleinsorge [5] uses an ILP model which is based on the model presented by Suhendra et al. [18]. We are using the exemplary program shown in Figure 5 to demonstrate the model. This model will be extended in Section 5.4 to enable bus-awareness.

Before the actual generation of the ILP model, the program needs to be analyzed twice in terms of WCET. Conventional WCET analyzers like AbsInt aiT [1] are not able to analyze bus-related latencies in a multicore setup, thus leading to an insufficient timing accuracy. However, analysis methods like those proposed by Kelter et al. [12] or Chattopadhyay et

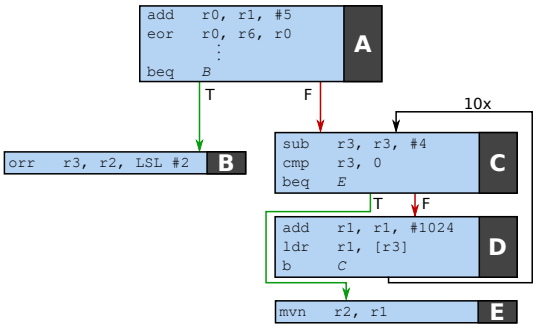

Figure 5 Exemplary CFG II.

al. [3] can be used to analyze multicore architectures including bus-related effects. The first analysis run is done with all basic blocks assigned to the Flash memory, while the second analysis is executed with all BBs assigned to a (virtually enlarged) SPM. The analysis runs are executed using multicore-enabled analyzing methods, so the analyzed timings include bus-related timings. The results of these analysis runs can be used to extract the net WCET per basic block ν, namely $C_{\nu,\text{Flash}}$ and $C_{\nu,\text{SPM}}$, denoting to which memory basic block ν was assigned to. Potential bus-related timings are included in the WCET of the basic blocks. Using these net WCETs, a timing gain G can be defined per basic block:

$$G_\nu = C_{\nu,\text{Flash}} - C_{\nu,\text{SPM}} \,. \tag{16}$$

G represents the timing profit in case a basic block is assigned to the SPM. Based on these timings, the WCET of the program can be modeled successively by introducing a variable w_ν which denotes the WCET of the path starting at basic block ν. We consider the timings of a block in terms of clock cycles, therefore integer variables are suitable to model the WCET of a basic block. The model is built up from the CFG's sink nodes.

$$w_B = C_{B,\text{Flash}} - x_B \cdot G_B \tag{17}$$
$$w_E = C_{E,\text{Flash}} - x_E \cdot G_E \tag{18}$$

x_ν is a Boolean decision variable and represents whether basic block ν is assigned to the SPM ($x_\nu = 1$) or not ($x_\nu = 0$). Subsequently, the control flow graph is traversed upwards. For each successor of a basic block, one individual constraint is added, containing its own net WCET and the corresponding successor's WCET. Regarding basic block A, this results in the following inequations:

$$w_A \geq C_{A,\text{Flash}} - x_A \cdot G_A + w_B \tag{19}$$
$$w_A \geq C_{A,\text{Flash}} - x_A \cdot G_A + w_{Loop} \tag{20}$$

The loop, which consists of basic blocks C and D, is modeled as a super-node. The partial WCET w_{Loop} starting at the entry of the loop is defined by its members, the loop bound and its successor.

$$w_{Loop} \geq c_{Loop} + C_{C,\text{Flash}} - x_C \cdot G_C + w_E \tag{21}$$
$$c_{Loop} \geq 10 \cdot w_{Entry} \tag{22}$$
$$w_{Entry} \geq C_{C,\text{Flash}} - x_C \cdot G_C + w_D \tag{23}$$
$$w_D \geq C_{D,\text{Flash}} - x_D \cdot G_D \tag{24}$$

$C_{C,\text{Flash}}$ is accounted for 11 times, since it is the head of the loop and is therefore executed one more time than the loop body. In order to restrict the number of basic blocks assigned to the scratchpad memory, an additional constraint has to be introduced:

$$S_{\text{SPM}} \geq x_A \cdot S_A + x_B \cdot S_B + x_C \cdot S_C + x_D \cdot S_D + x_E \cdot S_E \tag{25}$$

where S_{SPM} denotes the total size of the SPM and S_A the code size of basic block A. The overall WCET of the program can now be minimized by setting the objective function to minimize the WCET of the entry basic block, here A.

$$\min w_A \tag{26}$$

After assigning a block ν to the SPM, the control flow has to be repaired. This is required, since the program flow can be broken in several ways. In case the predecessor of ν did not contain an explicit jump to ν (e.g., see Figure 5: BB $A \rightarrow$ BB C), a new jump has to be introduced. However, even if an explicit jump was existing, it is likely to be not sufficient anymore, since the physical addresses of different memory regions presumably differ beyond the offset capabilities of a direct jump. Therefore, also the potential size and execution time of jump correction code has to be considered inside the ILP model using additional constraints. For the sake of simplicity and since this topic has already been discussed by Oehlert et al. [16], the additional ILP terms to consider such costs are omitted here. Nevertheless, the ILP model presented in this paper includes these constraints and considers their additional costs.

5.4 Bus-aware Extensions

As shown in Section 4, the WCETs extracted from the analysis runs of a BB are not safe anymore in case the current memory allocation of the program differs from the one used during analysis. More precisely, the WCET may differ if the temporal start of a basic block in regard to the bus schedule (the so-called *bus offset*) is different to the one analyzed. In regard to the program SPM allocation, this WCET change can be caused by two possibilities: If a basic block is allocated to the private SPM, but contains instructions which are explicitly accessing the shared memory, the bus offset during the access may be different, which may introduce additional waiting cycles. This is the case for data accessing instructions, since we assume the `.data` section to be placed inside the shared memory. In case the predecessor of a BB ν is assigned to the SPM (while ν resides in the shared memory), the bus offset at the execution start of ν may differ, which again may lead to a different WCET. Therefore, we extend the ILP model to predict these bus offsets and calculate bus-related penalties (or gains) based upon. We will retain the base model with its analyzed WCETs per block, but add these bus-related timing differences to it.

We assume all TDMA slots to be equally-sized and to be fixed to the length of exactly one Flash memory access delay F_{Flash}. This restriction enforces all possible accesses to the shared memory to be initiated during the first cycle of each core's bus slot. Thereby, all bus offsets at the beginning of a basic block placed inside the Flash are fixed and identical to the ones analysed. Accesses to the shared memory now serve as a kind of synchronization point. Since the bus offsets are known to be equal to the offsets during analysis, all WCETs of basic blocks placed inside the Flash can be safely extracted from the analysis again. In regard to the ARM7TDMI architecture used for evaluation purposes, this slot length restriction is acceptable in terms of timing since it fetches every instruction piecewise. Methods to relax this restriction while keeping the accuracy is part of our future work.

(a) Before sub basic block splitting **(b)** With sub basic block splitting applied

Figure 6 An exemplary BB containing an instruction with an explicit access to shared memory.

Additionally, all basic blocks which contain instructions with potential access to a shared memory are split up into sub basic blocks. These sub basic blocks either consist of *multiple* instructions which never access the shared data memory, or exactly *one* instruction which may then access the Flash. Example: An arbitrary BB Z is shown in Figure 6a. This basic block contains an instruction which accesses shared memory.

Figure 6b shows the sub basic blocks created from the former basic block Z. Sub basic block Z' now solely consists of the shared data memory accessing instruction.

This division of basic blocks is done in order to obtain the WCETs on a more detailed scale, since common WCET analyzers return the WCET per BB as the lowest granularity available. Using the timings of the sub basic blocks and the initial bus offset, the bus offset during the access can be predicted without further modifying existing analyzing techniques.

Bus Offset Calculation

For each sub basic block, the incoming and outgoing bus offsets are determined inside the ILP model. The incoming bus offset denotes the bus offset at the beginning of the execution of a basic block. In analogy to this, the outgoing bus offset describes the bus offset at the end of the execution of a basic block. Due to the different execution contexts of a basic block, it can have different execution times, varying between its Best-Case Execution Time (BCET) and its WCET. In this particular setup, these differing executions times are caused by pipeline effects or instructions with a possible varying execution time. Thus, the bus offset cannot be described as a scalar, but is rather an interval. This offset interval contains the lowest offset possible as well as the greatest. Inside the ILP model, an offset interval $\boldsymbol{o}$ is represented as two integer variables, o_{low} and o_{high}. The range of an offset variable like o_{low} or o_{high} is limited to the range of [0, P-1], where P is the total length of one bus period. In case of a wrap-around, i.e., $o_{\text{low}} > o_{\text{high}}$, the whole bus period is considered as a safe over-approximation. For every sub basic block ν, an offset interval $\boldsymbol{o}^{\text{In}}_{\nu}$ is added to the ILP model, representing the incoming bus offset interval of sub basic block ν. This offset interval is calculated as follows:

$$\boldsymbol{o}^{\text{In}}_{\nu} = \begin{cases} \boldsymbol{A}^{\text{In}}_{\nu,\text{Flash}} & \text{if } x_{\nu} = 0, \\ \bigcup_{\mu \in \text{Pred}(\nu)} \boldsymbol{o}^{\text{Out}}_{\mu,\nu,\text{W}} & \text{else.} \end{cases} \tag{27}$$

In case the Boolean SPM allocation variable x_{ν} is set to 0 (i.e., sub basic block ν would be assigned to the Flash memory), the incoming offset interval is equal to the interval extracted from the „all-in-Flash“ analysis. This is valid, since due to our restriction of slot lengths, we know that the actual execution start of a block placed inside the Flash memory can only happen at one single bus offset which we can extract from the analysis results. The potential

difference between the execution end of a preceding block and $\boldsymbol{o}_{\nu}^{\mathrm{In}}$ is later considered by adding a penalty timing to the preceding block.

If block ν is assigned to the SPM, $\boldsymbol{o}_{\nu}^{\mathrm{In}}$ has to be determined as the union of the outgoing bus intervals over all predecessors. The union over two given offset intervals $\boldsymbol{o_\nu}$ and $\boldsymbol{o_\mu}$ is defined using the following equations:

$$\boldsymbol{o_\nu} = (o_{\nu,\mathrm{low}}, o_{\nu,\mathrm{high}}) \tag{28}$$

$$\boldsymbol{o_\mu} = (o_{\mu,\mathrm{low}}, o_{\mu,\mathrm{high}}) \tag{29}$$

$$\boldsymbol{o_\nu} \cup \boldsymbol{o_\mu} = (\min(o_{\nu,\mathrm{low}}, o_{\mu,\mathrm{low}}), \max(o_{\nu,\mathrm{high}}, o_{\mu,\mathrm{high}})) \tag{30}$$

The min() and max() functions used in equation (30) are implemented as presented in Section 5.2. The interval $\boldsymbol{o}_{\mu,\nu,\mathrm{W}}^{\mathrm{Out}}$ represents the outgoing bus interval at sub basic block μ to successor ν, already including the effects of potential jump correction code.

Besides, for each successor of ν one offset interval $\boldsymbol{o}_{\nu,\mu,\mathrm{W}}^{\mathrm{Out}}$ has to be inserted into the ILP model (mind the switched indexes, since basic block ν is the source basic block in this case). This differentiation between successors is necessary, since e.g., a jump from the private SPM to the shared Flash memory requires an access to the bus to refill the pipeline, resulting in a contrasting bus offset in case the jump was not taken. It is defined as follows:

$$\boldsymbol{o}_{\nu,\mu,\mathrm{W}}^{\mathrm{Out}} = \begin{cases} \boldsymbol{o}_{\nu,\mathrm{WO}}^{\mathrm{Out}} & \text{if } x_\nu = x_\mu, \\ Q_{\mathrm{Flash}\to\mathrm{SPM}} & \text{if } x_\nu = \overline{x_\mu} = 0, \\ \boldsymbol{A}_{\mu,\mathrm{Flash}}^{\mathrm{In}} & \text{else.} \end{cases} \tag{31}$$

In case sub basic block ν and its successor μ are both placed inside the same memory, no additional jump correction is needed and the outgoing interval is identical to the offset interval not considering any jump correction, namely $\boldsymbol{o}_{\nu,\mathrm{WO}}^{\mathrm{Out}}$. If ν is placed inside the shared Flash memory and its successor μ in the SPM, an additional jump needs to be considered. Since the jump correction code will be placed inside the shared memory (where the ν resides), its bus offset at the end of the execution is always constant. This is due to the fact that the last instruction of a jump correction is always identical (an indirect jump). Because this instruction is placed inside the shared Flash memory, the fetching of it serves as a synchronization point. Therefore, the outgoing bus offset is constant under this circumstance and can be determined prior to the optimization. This offset is denoted as $Q_{\mathrm{Flash}\to\mathrm{SPM}}$.

In case sub basic block ν is assigned to the private SPM and its successor μ to the shared memory, a jump correction has to be done as well. Due to the fact that the successor is placed in the Flash memory, we can set the outgoing offset of ν to the analyzed incoming offset of μ (extracted from the „all-in-Flash“ analysis). During the execution of the final indirect jump as a part of the jump correction code, the processor needs to fetch the first instructions of block μ, which are placed in the shared memory. This is needed in order to refill the processor's pipeline, so the succeeding block does not start its execution with an empty pipeline (reminder: The ARM7TDMI architecture *always* assumes a jump not to be taken, so during the time the final indirect jump went into the execution phase, the pipeline was already filled with subsequent instructions from the SPM). Therefore, the outgoing offset of the jump correction code will be synchronized to the offset interval extracted from the analysis.

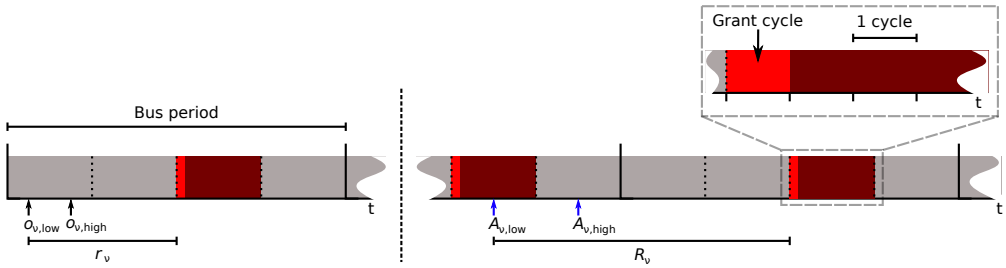

Figure 7 Two exemplary bus offset intervals $\boldsymbol{o}_\nu$ and $\boldsymbol{A}_\nu$ in regard to the bus schedule. The bus slot of the third core is highlighted, as well as the grant cycle during which an access can be issued.

Besides the outgoing bus offset interval including potential jump correction costs, a bus offset interval $\boldsymbol{o}^{\text{Out}}_{\nu,\text{WO}}$ is added to the ILP for each sub basic block ν, representing the outgoing bus offset without considering any jump correction. It can be determined as follows:

$$\boldsymbol{o}^{\text{Out}}_{\nu,\text{WO}} = \begin{cases} \boldsymbol{A}^{\text{Out}}_{\nu,\text{Flash}} & \text{if } x_\nu = 0, \\ \boldsymbol{A}^{\text{Out}}_{\nu,\text{SPM}} & \text{else if } H_\nu = 1, \\ (\boldsymbol{o}^{\text{In}}_\nu + \boldsymbol{T}_{\nu,\text{SPM}}) \bmod P & \text{else if } |\boldsymbol{T}_{\nu,\text{SPM}}| \leq P, \\ [0, P-1] & \text{else.} \end{cases} \tag{32}$$

In analogy to the incoming bus offset interval, the outgoing interval will be identical to the offset analyzed during the „all-in-Flash“ analysis run $\boldsymbol{A}^{\text{Out}}_{\nu,\text{Flash}}$ in case the sub basic block ν is placed inside the Flash memory. This is valid, since due to the slot lengths restriction we know that the bus offset during the start of the execution of block ν is fixed in this case. Therefore, its execution and also its outgoing bus offset interval will be always identical, independent from the temporal history.

If ν is assigned to the private SPM and has an explicit access to a shared memory region ($H_\nu = 1$), $\boldsymbol{o}^{\text{Out}}_{\nu,\text{WO}}$ will be set to the outgoing offset analysed during the „all-in-SPM“ analysis run. This is legitimate, since due to the sub basic block splitting, a sub basic block with a potential shared data memory access can only consist of this instruction itself. Therefore, the access can be regarded as a synchronization point, leading to the identical outgoing offset as resulted in the SPM analysis run.

Otherwise, the outgoing offset interval has to be calculated based on the incoming offset interval $\boldsymbol{o}^{\text{In}}_\nu$ and the analyzed execution time window (the difference between WCET and BCET) $\boldsymbol{T}_{\nu,\text{SPM}}$ when assigned to the SPM. However, in case the execution time window exceeds one whole bus period P, all possible bus offsets have to be considered.

Due to the nature of ILP, all possibilities of each ILP offset interval variable are calculated side by side and then chosen using a case distinction, implemented as presented in Section 5.2.

Bus-related Penalties

Based on the determined bus offset intervals per sub basic block, it is possible to predict the occurring timing related effects. For this purpose, we introduce two possible new ILP variables per sub basic block. The base penalty d_ν represents the additional cycles needed for the execution of a sub basic block ν due to an explicit access to a shared memory.

d_ν can also be negative if the different bus offset causes a better bus alignment in comparison to the previous analysis run. In this case, d_ν denotes a gain rather than a penalty. It is defined as follows:

$$d_\nu = \begin{cases} r_\nu^{\text{Data}} - R_\nu^{\text{Data}} & \text{if } x_\nu = H_\nu = 1, \\ 0 & \text{else.} \end{cases} \tag{33}$$

The ILP variable r_ν^{Data} describes the number of cycles needed at sub basic block ν to acquire a bus grant. It is required that ν is assigned to the private SPM and consists of an instruction which potentially accesses the shared memory through an explicit load/store instruction. Otherwise, r_ν^{Data} is always zero and does not need to be created.

Example: Figure 7 illustrates two exemplary bus offset intervals $\boldsymbol{o}_\nu$ and $\boldsymbol{A}_\nu$ in regard to the bus schedule. An arbitrary program is assumed to run on the third core of the system in its private SPM, possessing the third slot of each period which is highlighted. Since we restrict the initiation of a bus access to the first cycle of the corresponding slot, this cycle is highlighted in a lighter color as well. A sub basic block ν is assigned to the SPM and contains an access to the shared data memory. This access is tried to be issued at the shown bus offset interval $\boldsymbol{o}_\nu$. Considering the worst case, the processor has to stall at most r_ν cycles. Here, r_ν matches the ILP variable r_ν^{Data}, representing the greatest number of stalling cycles considering the current program allocation, until the bus grant is received.

The second offset interval $\boldsymbol{A}_\nu$ shown in the figure represents the bus offset interval extracted from the WCET analysis at the same sub basic block ν. Therefore, R_ν resembles the constant R_ν^{Data} from Equation (33), namely the number of cycles accounted by the WCET analyzer to gain the bus grant. Since the actual memory access delay is identical during both executions and already accounted, only the difference in stalling cycles until the bus grant is relevant to calculate. For this reason, the difference between r_ν^{Data} and R_ν^{Data} is calculated. Regarding Figure 7, the ILP-chosen allocation of blocks leads to a lower number of stall cycles needed at sub basic block ν in comparison to the „all-in-SPM“ allocation, since r_ν^{Data} is lower than R_ν^{Data}.

In order to define r_ν^{Data}, the number of cycles between the bus offset interval $\boldsymbol{o}_\nu^{\text{In}}$ and the next granted bus slot of the corresponding core is calculated.

$$a_{\text{low}} = \left(I \cdot F_{\text{Flash}} - o_{\nu,\text{low}}^{\text{In}}\right) \bmod P \tag{34}$$

$$a_{\text{high}} = \left(I \cdot F_{\text{Flash}} - o_{\nu,\text{high}}^{\text{In}}\right) \bmod P \tag{35}$$

$$r_\nu^{\text{Data}} = \begin{cases} P - 1 & \text{if } o_{\nu,\text{low}} \leq I \cdot F_{\text{Flash}} + 1 \leq o_{\nu,\text{high}}, \\ \max(a_{\text{low}}, a_{\text{high}}) & \text{else.} \end{cases} \tag{36}$$

The term $I \cdot F_{\text{Flash}}$ equals the bus offset of a core I at which a bus access is granted. The constant R_ν^{Data} is calculated in the same manner, but using the bus offsets extracted from the „all-in-SPM“ analysis run.

The second ILP variable introduced per sub basic block ν is $l_{\nu,\mu}$. This variable represents the potential additional bus waiting cycles during the execution of jump correction code. It is defined as follows:

$$l_{\nu,\mu} = \begin{cases} r_\nu^{\text{Jump}} & \text{if } x_\nu = \overline{x_\mu} = 1, \\ 0 & \text{else.} \end{cases} \tag{37}$$

Example: A sub basic block ν is assigned to the private SPM of the third core of a system, while its successor μ resides in the shared Flash memory. This circumstance requires the

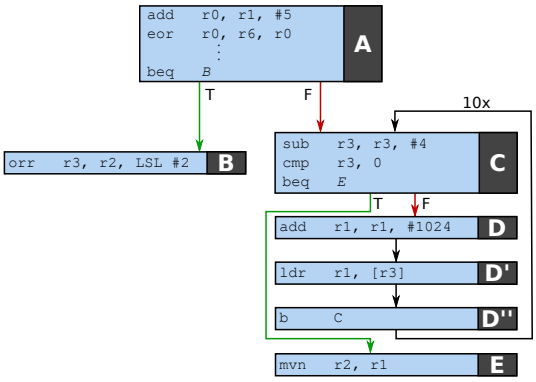

Figure 8 Exemplary CFG from Figure 5 with sub basic block splitting applied.

introduction of jump correction code subsequent to ν. During the end of the execution of the appended jump code, the first instructions of the succeeding block μ have to be fetched, so μ does not start with an empty pipeline. Since block μ is placed inside the shared Flash memory, the jump correction code will cause an additional bus access during the fetch of its leading instructions. Referring to Figure 7, $\boldsymbol{o}_\nu$ resembles the offset interval of sub basic block ν when such an access is tried to be issued. The variable r_ν^{Jump} then describes the greatest number of cycles which are needed, based on offset interval $\boldsymbol{o}_{\nu,\text{WO}}^{\text{Out}}$, to reach a valid bus slot.

The variable r_ν^{Jump} is determined in the same fashion as r_ν^{Data}, but utilizing the offset interval $\boldsymbol{o}_{\nu,\text{WO}}^{\text{Out}}$. Since these timing costs did not exist in the initial analysis runs, no subtractive term as in Equation (33) is added. All other timings of jump correction code are constant and can be calculated upfront. Those timings are already considered inside the jump correction costs of the base model.

Final ILP Model

The presented additions are integrated into the base ILP model. Prior to the initial WCET analysis runs, the sub base block splitting is applied.

Regarding the exemplary control flow graph shown in Figure 5, the additional variables d and l are added to the WCET constraints of the corresponding sub basic blocks. The CFG with splitting applied is shown in Figure 8.

$$w_A \geq C_{A,\text{Flash}} - x_A \cdot G_A + w_B + l_{A,B} \tag{38}$$
$$w_A \geq C_{A,\text{Flash}} - x_A \cdot G_A + w_{Loop} + l_{A,Loop} \tag{39}$$
$$w_B = C_{B,\text{Flash}} - x_B \cdot G_B \tag{40}$$
$$w_{Loop} \geq c_{Loop} + C_{C,\text{Flash}} - x_C \cdot G_C + w_E + l_{C,E} \tag{41}$$
$$c_{Loop} \geq 10 \cdot w_{Entry} \tag{42}$$
$$w_{Entry} \geq C_{C,\text{Flash}} - x_C \cdot G_C + w_D + l_{C,D} \tag{43}$$
$$w_D \geq C_{D,\text{Flash}} - x_D \cdot G_D + w_{D'} + l_{D,D'} \tag{44}$$
$$w_{D'} \geq C_{D',\text{Flash}} - x_{D'} \cdot G_{D'} + w_{D''} + l_{D',D''} + d_{D'} \tag{45}$$
$$w_{D''} \geq C_{D'',\text{Flash}} - x_{D''} \cdot G_{D''} + l_{D'',C} \tag{46}$$
$$w_E = C_{E,\text{Flash}} - x_E \cdot G_E \tag{47}$$
$$S_{\text{SPM}} \geq x_A \cdot S_A + x_B \cdot S_B + x_C \cdot S_C + ... + x_E \cdot S_E \tag{48}$$

The data access penalty d is only introduced to sub basic block D', since it is the only block with a potential access to a shared memory region. The objective function is kept from the base ILP. The constraints to consider the additional spatial costs of possible jump correction code are derived from the base ILP as well, yet not shown here to avoid unnecessary complications.

6 Evolutionary Algorithm

This section describes the genetic algorithm used as a reference for the ILP-based static bus-aware multicore SPM allocation optimization. The optimization is a classical genetic algorithm as described by Goldberg [7]. It starts with a set of individuals of which each holds a set of binary decision variables $x_{\nu,I}$ denoting whether basic block ν of core I will be assigned to SPM. We assume that there is no shared code between the cores.

Unless stated otherwise, a random selection is drawn from a uniform distribution. We create the initial set of N_{Ind} individuals as follows:

- The first individual is left with all basic blocks in Flash memory.
- For all other individuals, we virtually assign all blocks to the SPM and then randomly remove basic blocks of each core from SPM until the SPM memory is no longer overflowing.

For recombination of two individuals A and B, our tests showed good results with a simple one-point recombination with multi-bit mutation. We first randomly determine the core I to be crossed over. For the selected core, we randomly determine the position i at which the two individuals will be merged.

The new individual C will have the following new assignment:

$$C = A[0, i-1] \mid B[i, N_{B,I} - 1] \tag{49}$$

$N_{B,I}$ denotes the total amount of basic blocks contained by the task allocated to core I. The first i decision variables of the new individual C will be taken from individual A, while the second part is taken from individual B. Subsequently we randomly choose a number of maximum mutations M for the SPM assignment in the crossed over core I.

We then randomly select M basic blocks to mutate. Whether or not the assignments of these randomly selected basic blocks will be toggled is then again randomly determined for each with a user-definable probability.

Using the allocation determined by the new individual C, a jump correction is performed in order to repair the control flow graph. If this final SPM assignment including the inserted jump correction code fits into the physically available SPM, no repair is necessary. If this is not the case, either the number of basic blocks assigned to the SPM was too high, or the jump correction code overfilled the SPM boundaries. In this case, we again randomly remove blocks from the SPM and perform a jump correction respectively until the assignment is valid.

Finally, the new individual is analyzed using the WCET analyzing methods proposed by Kelter [11] to assess the new WCET. Because the WCET analysis automatically analyzes bus penalties and accounts for them in the task's worst execution timing behavior, the genetic optimization is inherently bus-aware.

For the next generation, the N_{Ind} fittest individuals are selected. The allocation of a program on one core does not interfere with the execution of a program on another core due to the TDMA schedule with fixed slot lengths. As a result, our fitness function can simply be chosen to minimize the sum over the WCETs of all tasks.

The optimization terminates if the WCET reduction of any core over 2 generations is smaller than a user-definable threshold ϵ, or a user-definable amount of time has gone by.

7 Evaluation

The presented bus-aware ILP-based instruction SPM allocation and the one based on evolutionary algorithms were implemented for a multicore ARM7TDMI architecture described in Section 3. We use the resulting WCET of the bus-unaware ILP-based instruction SPM allocation (described in Section 5) as a baseline. The access delay of the private SPM is assumed to be 1 cycle, while the access delay of the shared Flash memory is assumed as 6 cycles in case the bus grant is acquired. In reference to the bus slot length restriction described in Section 5.4, the bus schedule consists of equally-sized bus slots. The length of a slot is set to the access delay of the Flash memory F_{Flash} (6 cycles). The (sub) basic blocks assigned to the scratchpad memory are loaded into the memory prior to the actual execution of the program. Therefore, the initial cycles required to transfer the corresponding blocks into the SPM do not need to be considered in terms of the WCET of a program.

The MRTC benchmark suite [8] was used for evaluation purposes with annotated loop bounds from the TACLeBench project [4]. The `duff` benchmark was excluded from the set of benchmarks, since it contains an irregular loop which can not be modeled using our current ILP models. Besides, the benchmarks `petrinet` and `statemate` were excluded due to timeouts (execution time > 15h) during the bus-aware ILP-based optimization. However, the evolutionary approach was able to terminate in the given time limit for these benchmarks. The WCET analyses for the ARM7TDMI multicore platform were done by using methods described by Kelter [11].

For the instruction SPM allocation based on evolutionary algorithms, the parameters were carefully chosen to compromise between execution time and effectiveness:

- Initial population: 20
- Number of parents per generation: 20
- Number of offspring individuals: 20
- Maximum number of generations: 50
- Mutation probability: 0.2
- Multibit mutation
- Single-point crossover

All evaluations were performed on an Intel Xeon Server. ILPs were solved using Gurobi 7.0.1 using 20 threads. All benchmarks were compiled with the WCET-aware C compiler (WCC) [6] and the `-O2` flag applied which enables several ACET-oriented compiler optimizations. The private scratchpad memory size is set individually for each benchmark, adjusting it to 50% relative to the benchmark's code size. All optimizations were performed for a dualcore, quadcore and octacore target platform.

Figure 9 shows the WCET of each benchmark optimized using the bus-aware ILP-based allocation and based on evolutionary algorithms, relative to the optimized WCET using the bus-unaware ILP-based optimization. The programs were executed on one core of the dualcore platform. Since we are using a TDMA scheduling policy for the bus, it is irrelevant under which combinations the benchmarks are executed. The execution of a benchmark on one core does not influence the WCET of a benchmark being executed on another core, since the bus slot lengths are fixed for each platform. On average (geometric mean), the bus-aware ILP-based optimization results in a WCET 23% lower than the unaware ILP-based optimization. The benchmark `crc` yields the greatest reduction in terms of WCET with

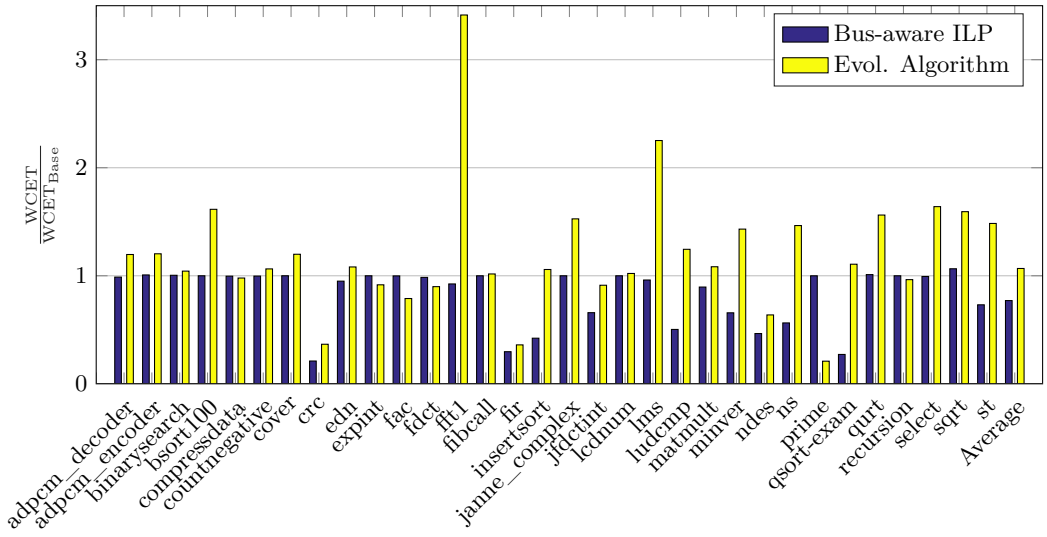

Figure 9 Relative optimized WCETs using the dualcore ARM7TDMI platform.

79%, while `sqrt` results in a 6% worse WCET. The bus-aware ILP-based optimization can lead to a worse allocation in comparison to the unaware model in certain cases, since the model potentially allocates blocks in a pessimistic way due to uncertainties related to bus effects. Because the bus-unaware ILP model does not consider potential bus effects, it can result in a lower WCET in case the bus-aware model is overly pessimistic.

The instruction SPM allocation based on evolutionary algorithms returns on average a WCET which is 7% greater than the baseline WCET. The benchmark with the greatest WCET reduction is `prime` with 79%, for which the bus-aware ILP optimization did not return any noticeable WCET reduction. This is mostly caused by the fact that the ILP model is not able to represent execution contexts, which leads to a harsh over-approximation regarding the total WCET. This behavior of the benchmark `prime` can be observed independent from the number of cores per system. On the contrary, the evolutionary algorithm is able to evaluate these effects inherently, since the used WCET analyzing methods consider execution contexts. However, the benchmark `fft1` results in a WCET which is 241% greater than the baseline WCET.

Figure 10 shows the results of the same experiment setup, but using a quadcore platform. The results resemble the experiment on a dualcore platform, yet the extremes are pushed further to either direction. In case of the bus-aware ILP-based optimization, still the `crc` benchmark results in the best reduction with 86%. Yet again, `sqrt` represents the worst result in this configuration with a 5% greater WCET. The bus-aware ILP optimization results in a 28% lower timing in comparison to the baseline WCET on average per benchmark when performed on a quadcore system.

On average, the evolutionary-based optimization leads to worse results in comparison to the dualcore experiment with an average WCET 12% higher than the baseline. With a WCET reduction of 72% compared against the baseline, the benchmark `fir` provides the best result using the evolutionary approach on a quadcore platform. Same as in the dualcore experiment, the `fft1` benchmark returns the worst result also in this configuration with a WCET 437% greater than the baseline WCET.

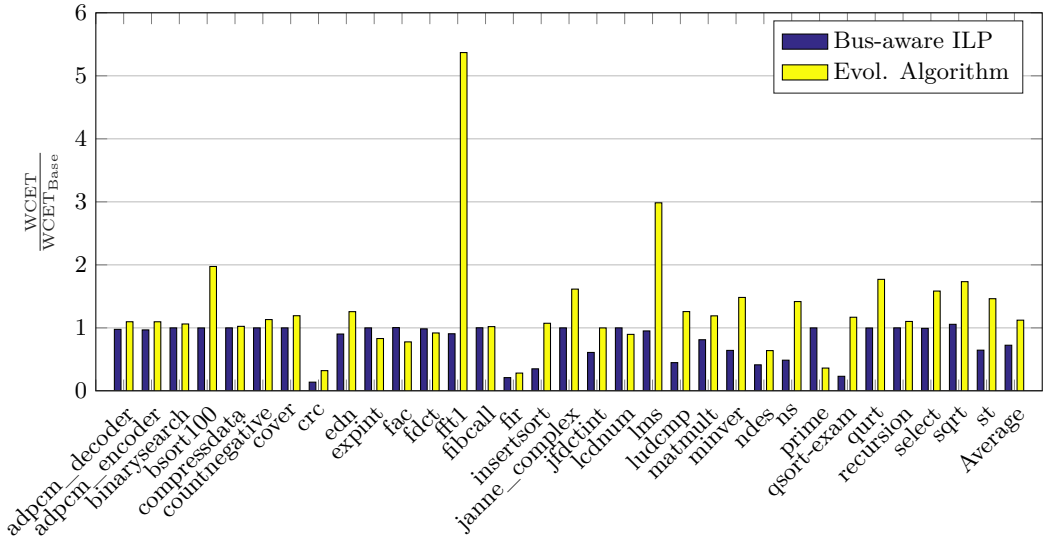

Figure 10 Relative optimized WCETs using the quadcore ARM7TDMI platform.

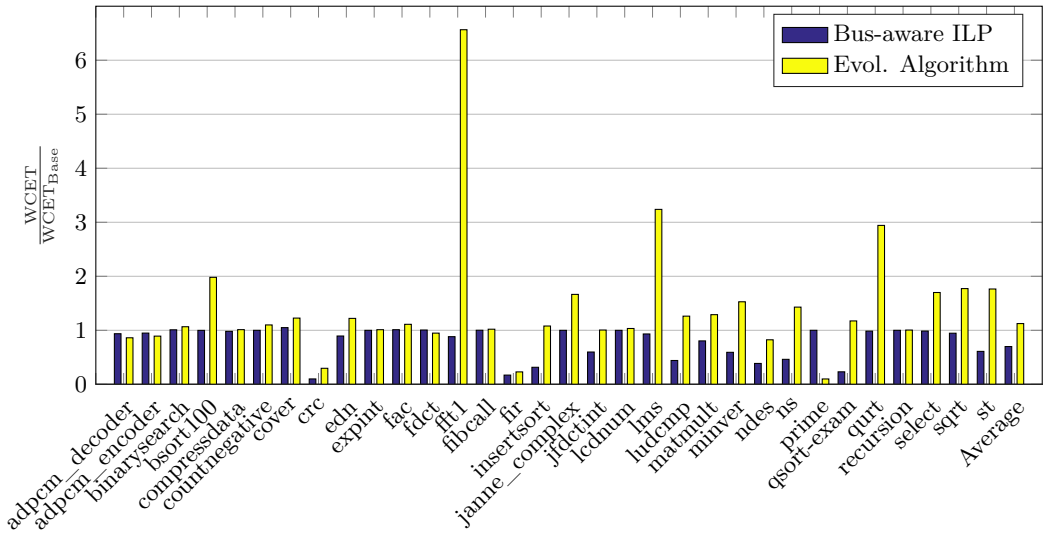

Figure 11 Relative optimized WCETs using the octacore ARM7TDMI platform.

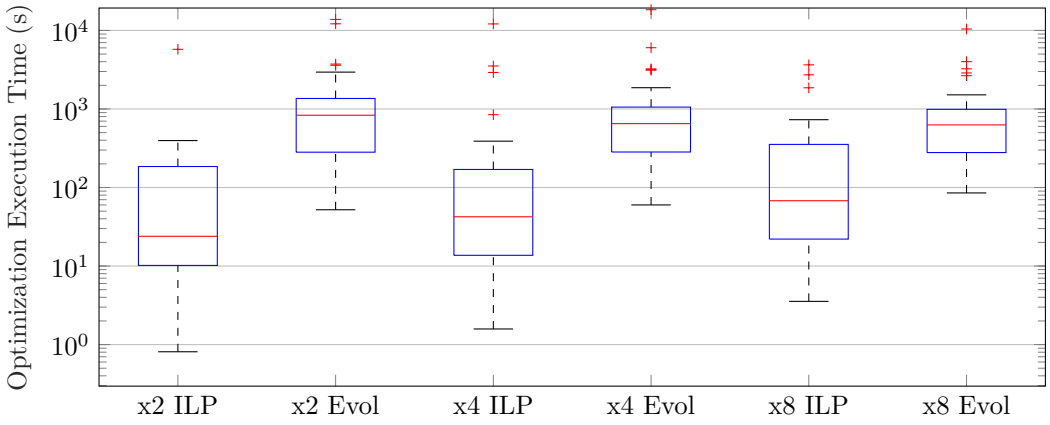

Figure 12 Overall Execution Time of the Optimizations for different Platforms.

In Figure 11, the results extracted from the experiments performed on an octacore ARM7TDMI platform are shown. In general, the results resemble a similar pattern as seen in the previous experiments. The bus-aware ILP-based optimization results in average a WCET which is 30% lower than the baseline WCET. The benchmark `crc` yields the lowest WCET in comparison to the bus-unaware ILP optimization for the octacore platform with a 90% lower WCET, while `cover` results in a 5% higher WCET.

The average WCET per benchmark using the evolutionary-based instruction SPM allocation is 12% higher in comparison to the baseline WCET. The greatest timing reduction using the evolutionary-based optimization in the octacore configuration is achieved for the `prime` benchmark with a 90% lower WCET in comparison to the bus-unaware ILP optimization, while `fft1` results in a 556% higher WCET.

Overall, it is observable that the advantage of the bus-aware ILP optimization in comparison to its unaware opponent rises with the number of cores in the system. This conclusion is expectable, since the impact of bus-related effects also increases with the number of cores, since the possible stalling times rise with a longer bus period. Meanwhile, the evolutionary-based optimization's quality degrades with an increasing number of cores. This is likely to be the case, since the penalty induced by only *one* badly allocated basic block heavily increases with an increasing number of cores, due to greater stalling times. Therefore, the evolutionary algorithm requires a larger number of generations to reach a proper allocation. Since we set a fixed upper bound of the maximum generations, it will get more likely that the optimization will be canceled before an adequate allocation is reached with an increasing number of cores.

Figure 12 shows the overall execution times of the bus-aware ILP-based instruction SPM allocation and the evolutionary-based approach, separated according to the number of cores used in the platform.

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. Execution times outside the region between the whiskers are depicted with a „+“-symbol. It is noticeable that independent from the number of cores used inside the system, the execution time of the ILP-based optimization is distinctively lower in terms of the median in comparison the evolutionary-based optimization. This is likely to be caused by the plenty of analysis runs required by the evolutionary algorithm, while the ILP-based approach only relies on two runs. The time required by the evolutionary algorithm could be decreased by decreasing the number of maximum generations or the

number of individuals per generation. However this would likely lead to a decreased quality of optimization. Furthermore, the evolutionary approach's timing can be still improved by enabling parallelism, which is yet to be implemented.

8 Conclusion and Future Work

We showed a precise bus-aware ILP-based instruction scratchpad memory allocation to reduce the worst-case execution time of a program. This approach includes the dynamic prediction of bus offsets and their resulting timing effects inside the ILP model. We showed, that using this optimization, it is possible to reduce the WCET of a program in comparison to a bus-unaware ILP-based optimization by on average 26%, with an average runtime significantly lower than the genetic approach. On the downside, the approach heavily increases the complexity of the underlying model, especially for data intensive programs.

Besides, we showed a first approach based on evolutionary algorithms for instruction SPM allocation in multicore platforms which considers the timing effects of the bus architecture. Using this approach, the experiments returned a WCET reduction up to 90% in comparison the a bus-unaware ILP-based optimization.

As a part of future work, we plan to consider caches in our bus-aware ILP model. Integrating the results of cache analyses could greatly improve the WCET of a program furthermore, since less bus accesses would be required.

Further we plan to relax the bus slot length restriction discussed in this paper and expand the model to data scratchpad memory allocation.

Besides, we plan to further fine-tune our approach based on evolutionary algorithms to speed up the convergence to near-optimal results. Therefore, we intend to introduce parallelism during the fitness calculation of the individuals.

References

1 AbsInt Angewandte Informatik, GmbH. aiT Worst-Case Execution Time Analyzers, 2017.

2 Sudipta Chattopadhyay and Abhik Roychoudhury. Static Bus Schedule Aware Scratchpad Allocation in Multiprocessors. In *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES'11, pages 11–20, New York, NY, USA, 2011. ACM. doi:10.1145/1967677.1967680.

3 Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling Shared Cache and Bus in Multi-cores for Timing Analysis. In *Proceedings of the 13th International Workshop on Software and Compilers for Embedded Systems*, SCOPES'10, pages 6:1–6:10, New York, NY, USA, 2010. ACM. doi:10.1145/1811212.1811220.

4 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET)*, OpenAccess Series in Informatics (OASIcs), pages 2:1–2:10, Toulouse, France, 2016. doi:10.4230/OASIcs.WCET.2016.2.

5 Heiko Falk and Jan C. Kleinsorge. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *Proceedings of the 46th Annual Design Automation Conference*, DAC, pages 732–737, San Francisco, CA, USA, 2009. doi:10.1145/1629911.1630101.

6 Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, 46(2):251–300, 2010. doi:10.1007/s11241-010-9101-x.

7 David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc., Boston, MA, USA, 1989.

8 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, OASIcs, pages 136–146, Dagstuhl, Germany, 2010. doi:10.4230/OASIcs.WCET.2010.136.

9 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):12:1–12:25, May 2015. doi:10.1145/2738051.

10 T. Kelter, H. Borghorst, and P. Marwedel. WCET-aware scheduling optimizations for multi-core real-time systems. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 67–74, Samos, Greece, July 2014. doi:10.1109/SAMOS.2014.6893196.

11 Timon Kelter. *WCET Analysis and Optimization for Multi-Core Real-Time Systems*. PhD thesis, TU Dortmund, Department of Computer Science, Dortmund, Germany, March 2015.

12 Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems*, 50(2):185–229, 2014. doi:10.1007/s11241-013-9189-x.

13 Y. Kim, D. Broman, J. Cai, and A. Shrivastaval. WCET-aware dynamic code management on scratchpads for Software-Managed Multicores. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 179–188, Berlin, Germany, April 2014. doi:10.1109/RTAS.2014.6926001.

14 Donald Ervin Knuth. *Fundamental algorithms*. The art of computer programming. Addison-Wesley, Reading, MA, USA, 3. ed edition, 1997.

15 Yu Liu and Wei Zhang. Scratchpad Memory Architectures and Allocation Algorithms for Hard Real-Time Multicore Processors. *Journal of Computing Science and Engineering*, 9(2):51–72, 2015. doi:10.5626/JCSE.2015.9.2.51.

16 Dominic Oehlert, Arno Luppold, and Heiko Falk. Practical Challenges of ILP-based SPM Allocation Optimizations. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, SCOPES'16, pages 86–89, New York, NY, USA, 2016. ACM. doi:10.1145/2906363.2906371.

17 Vivy Suhendra and Tulika Mitra. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-cores. In *Proceedings of the 45th Annual Design Automation Conference*, DAC'08, pages 300–303, New York, NY, USA, 2008. ACM. doi:10.1145/1391469.1391545.

18 Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, RTSS'05, pages 223–232, Washington, DC, USA, 2005. IEEE Computer Society. URL: 10.1109/RTSS.2005.45.