Compiling for the Worst Case: Memory Allocation for Multi-task and Multi-core Hard Real-time Systems

ARNO LUPPOLD, DOMINIC OEHLERT, and HEIKO FALK, Hamburg University of Technology, Germany

Modern embedded hard real-time systems feature multiple tasks running on multiple processing cores. Schedulability analysis of such systems is usually performed on an abstract system level with each task being represented as a black box with fixed timing properties. If timing constraints are violated, then optimizing the system on a code-level to achieve schedulability is a tedious task.

To tackle this issue, we propose an extension to the WCET-aware C Compiler framework WCC. We integrated an optimization framework based on Integer-Linear Programming into the WCC that is able to optimize a multi-core system with multiple tasks running on each core with regards to its schedulability. We evaluate the framework by providing two approaches on a schedulability aware static Scratchpad Memory (SPM) allocation: one based on Integer-Linear Programming (ILP) and one based on a genetic algorithm.

CCS Concepts: • **Mathematics of computing** → **Solvers**; *Integer programming*; • **Computer systems organization**→ **Embedded software**; **Real-time systems**; Embedded and cyber-physical systems; • **Software and its engineering**→ **Compilers**; • **Theory of computation**→ *Integer programming*;

Additional Key Words and Phrases: WCET optimization, multi-core systems, scheduling analysis

ACM Reference format:
Arno Luppold, Dominic Oehlert, and Heiko Falk. 2020. Compiling for the Worst Case: Memory Allocation for Multi-task and Multi-core Hard Real-time Systems. *ACM Trans. Embed. Comput. Syst.* 19, 2, Article 14 (March 2020), 26 pages.
https://doi.org/10.1145/3381752

1 INTRODUCTION

An embedded system that has to comply to predetermined timing constraints is called a "real-time system." If violating a timing constraint even once may lead to catastrophic failure of the underlying system, then the embedded system is said to be a "hard" real-time system. Modern embedded hard real-time systems are full-fledged preemptive multi-task systems running on complex multi-core micro-controllers. To analyze such systems' timing behavior, the Worst-Case Execution Time (WCET) of each task is usually analyzed in a stand-alone fashion and then passed on to

This work received funding from Deutsche Forschungsgemeinschaft (DFG) under grant FA 1017/3-1. This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 779882.
Authors' addresses: A. Luppold, D. Oehlert, and H. Falk, Hamburg University of Technology, Am Schwarzenberg-Campus 3, Hamburg, 21073, Germany; emails: {arno.luppold, dominic.oehlert, heiko.falk}@tuhh.de.

1539-9087/2020/03-ART14
https://doi.org/10.1145/3381752

system-level schedulability analyses. These analyses will then safely predict whether each task provably meets its timing constraints, even if it is blocked or interrupted by other tasks. Over the past decades, schedulability analyses have made huge progress and are able to predict the worst-case behavior of arbitrarily triggered systems. Unfortunately, up to this point, compiler-based optimization techniques have not kept up with this pace. In a multi-task and multi-core system, there is no longer *the* WCET to optimize. Instead, when optimizing one task's WCET, both positive and negative side effects on other tasks' runtime behavior may occur.

This article presents two different approaches that tackle this issue by extending the WCET-aware C Compiler WCC: First, we present an ILP-based approach that is able to deterministically provide an optimal solution within the bounds of the underlying formal model. Then, we extend the genetic optimization from Oehlert et al. [37] to be able to compare the ILP approach in terms of optimization speed and quality. Due to the high optimization potential of so-called Scratchpad Memories (SPM), which are very fast but also very small local memories, we illustrate the potential of these new approaches by exemplarily using a static instruction SPM allocation. The key contributions of this article are:

- We investigate schedulability-aware memory allocation for multi-task *and* multi-core systems using an ILP-based and a genetic approach.
- We show that worst-case timing analysis and optimization are simplified by assuming that one bus slot length equals the access time of one memory access. We further show that this assumption does not lead to a negative average-case performance of this system.
- We propose two new fitness measures to compare the quality of two unschedulable systems for the gentic algorithm.
- A previously published approach on schedulability-aware ILP optimization [29] is improved such that a quadratic growth of the ILP model is reduced to linear growth only.
- We show that the ILP-based approach is able to significantly improve the schedulability of multi-task and multi-core systems within reasonable optimization time.

This article is organized as follows: Section 2 gives an overview of previously existing approaches. Section 3 explains the system model and gives preliminary explanations. Section 4 gives an overview over commonly used symbols and briefly introduces an existing ILP-based single-task static instruction SPM allocation that is integrated into WCC. Section 5 continues by elaborating on how to account for the effects of shared buses on an ARM7TDMI-based multi-core platform. Section 6 further extends the ILP formulations to account for task activation patterns and the used scheduling algorithm. This allows for a combined multi-task and multi-core aware optimization. Section 7 introduces a genetic algorithm as alternative to the ILP-based framework. Both approaches are evaluated in Section 8. A conclusion and a brief discussion of future challenges close the article.

2 RELATED WORK

A multi-tasking hard real-time system consists of multiple tasks running on one or multiple computational units. The analysis of strictly periodical hard real-time systems running on one single processing unit starts with Liu and Layland [24] in the early '70s and Joseph et al. [18]. A system is called *schedulable* if it will provably meet all timing requirements even under worst-case circumstances. Modern analysis techniques are now based on event-driven systems [13, 39, 44]. These techniques are able to analyze arbitrarily triggered systems distributed over multiple processing units. The techniques lead to high-level schedulability analysis tools like the so-called Real-Time Calculus [44] or the commercially available SymTA/S [17]. Despite the power of these techniques and tools, they all consider tasks as black boxes with fixed timing properties. If the analysis proves

the system to be not schedulable, i.e., at least one task might miss its deadline, then such approaches are not able to give any hint on how to modify the system to actually fix the problem.

On a still high abstraction level, *sensitivity analysis* tries to fill this gap [4, 34, 46]. These approaches try and give hints to the system designer which timing properties could be modified to achieve schedulability. However, these approaches are also working at a high abstraction layer, unaware of any micro-architectural details of the underlying platform and the actual code of the tasks. As a result, any proposed changes to the worst-case timing behavior of tasks are purely speculative. The sensitivity analysis does not know whether and how a proposed modification can actually be achieved without expensive hardware modifications.

To provide a good basis for any code-level optimization as well as the system-level analysis, the WCET of each task has to be analyzed. Unfortunately, the worst-case runtime of an arbitrary program cannot be safely predicted, as this would imply solving the halting problem [43]. To overcome this issue, any task in a hard real-time system is subject to several restrictions. Most importantly, the system must be designed such that safe upper bounds on any dynamic component such as loops or recursive function calls can be given at system design time. This so-called loop bound analysis can be either performed manually or automated [8, 16, 25]. Using this information and detailed knowledge about the target hardware, WCET analyzers such as Chronos [23] or the commercial AbsInt aiT [1] can be used to give safe over-approximations on the WCET of a given task.

In the past, compiler-based WCET *optimization* techniques have proven to be able to drastically improve on the worst-case runtime behavior of a hard real-time system [10, 26, 41]. Over the past years, multiple WCET-aware code optimizations were integrated into one common C compiler framework: the WCET-aware C Compiler WCC [11]. However, in the past, all these optimizations focused on optimizing single-task systems running isolated on one single processing core. This means that they could only optimize *the* WCET of a task.

Multi-core specific effects like conflicts during shared memory/bus accesses were not modeled. Kelter et al. [20] started to tackle multi-core specific effects like conflicts during shared memory/bus accesses. A holistic multi-core bus-aware WCET analysis framework was proposed in References [19, 21]. It allows for the analysis of multi-core sytems connected by a bus using either priority or time-arbitrated access or combinations of both. Oehlert et al. [35] showed that the analysis framework within WCC can be used for a micro-architectural WCET analysis with shared data or instruction accesses over commonly used buses like, e.g., FlexRay. Based on this analysis framework, a static instruction SPM optimization that schedules bus accesses to a TDMA bus such that interference delays are minimized on each core was proposed [37]. These approaches are able to calculate and optimize the WCET of a task considering multi-core effects like a shared memory bus. However, to cope with the inherent complexity of these analyses, multi-core support was mostly restricted to the use-case of one dedicated task running on each core. The optimization cannot account for scheduling algorithms and subsequent task preemptions on each core.

In case of multiple tasks, the optimization has no sound model guiding it on which task should be optimized to which extent to produce an optimal effect on the schedulability of the overall system. Optimization of such systems at the compiler level proved to be promising but also to be a computationally highly complex optimization problem. Recently, we proposed an ILP-based SPM allocation optimization that specifically places basic blocks of a task into the fast-but-small SPM memory such that a previously unschedulable system becomes schedulable. This is done for both strictly periodical systems [28] as well as for arbitrary event-based systems [29]. The drawback of these approaches is that, up to now, they in turn did not cover multi-core behavior.

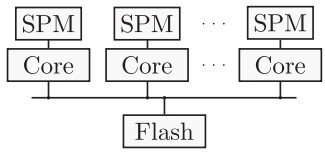

Fig. 1. Multi-core architecture used within this article.

3 SYSTEM MODEL AND PRELIMINARIES

For our optimization framework, we make the following assumptions:

Task Mapping. Task mapping to cores is considered to be fixed. All tasks have already been mapped to a core prior to applying the optimization framework. While task to core mapping is certainly a relevant problem, it is beyond the scope of this article.

Independent Tasks. Currently, we only consider independently running tasks without any inter-task dependencies.

Homogeneity. We assume that all cores are homogeneous. While this is not a real limitation of our analysis and optimization framework, it heavily simplifies notations and evaluation.

Caches. This approach does not yet consider caches. While caches start being used in the domain of hard real-time systems, they are still often deactivated to provide for tighter analysis estimates.

Time Units. We assume that all time units are expressed as integer values given in a multiple of CPU cycles. This does not impose any restriction.

Flow Facts. Generating flow facts that describe maximum loop iterations and recursion depths has already been researched intensively. We therefore assume that any flow facts have been calculated previously and are annotated to the source code. Our compiler framework WCC is then able to automatically exploit them for WCET analyses and optimizations [11].

SPM Memory. This work focuses on *instruction* SPM allocations. It is assumed that all data reside in a global shared memory. Bus access latencies due to data accesses are considered in any WCET *analysis* performed in this article. The upcoming ILP *optimization model* does not explicitly model these penalties. The evaluation results will show that this simplification can be made while still getting very good optimization results.

3.1 Memory Layout

Figure 1 shows the system setup used within this article. Each core has a private SPM. All cores are connected via a TDMA-scheduled bus. Additionally, a shared Flash memory is connected to the bus. The bus decides which core can access the shared bus at which given point in time based on a time schedule. The so-called TDMA schedule consists of several time slots and is repeated constantly. Each of these time slots is associated to one of the cores in which it can exclusively access the bus. Since the behavior of a single core does not influence the overall bus arbitration scheme (as the TDMA schedule is fixed), upper timing bounds for a bus access can be trivially derived.

In our system model, each core has exactly one dedicated time slot inside the TDMA schedule in which it can exclusively access the bus. All TDMA slots are set to a fixed and equal length. Moreover, we assume that a slot length is set to the latency of one Flash memory access, such that exactly one access to the shared Flash can be done when initiated at the very beginning of a slot.

Though advocated for the use in hard real-time systems [7] and supported by popular on-chip buses such as, e.g., AMBA [33], many COTS multi-core architectures do not support a TDMA

bus arbitration policy natively. As a possible solution for architectures not supported, Ziccardi et al. [47] demonstrated an approach to enforce a configurable TDMA bus arbitration by software means on an AURIX TC277TU multi-core processor.

3.2 Task Model

To analyze whether all task sets on all cores will provably meet their deadlines, a formal task model must be introduced. At a system's level of abstraction, we describe a task τ_i by the following tuple:

$$\tau_i = (c_i, D_i, \eta_i(\Delta t)), \tag{1}$$

where i is a non-negative integer value that uniquely identifies the task. For fixed-priority scheduling algorithms, we assume that i also denotes a task's priority. We define that a numerically lower value denotes a higher priority; i.e., task τ_0 has the highest priority in the system. For dynamic-priority scheduling schemes like EDF, i is merely used as an index to identify a certain task.

c_i is the Worst-Case Execution Time (WCET) of the task. This denotes the maximum time the task needs for its execution if it was executed in a stand-alone manner. D_i is the deadline of the task relative to its activation in the system. In a real system, this deadline is defined by the physical process with which the embedded system is interfering and may not be modified.

Finally, $\eta_i(\Delta t)$ is the so-called density function. It describes the maximum number of events that trigger an execution of τ_i within a given time interval Δt. For $\Delta t < 0$, the number of events in a negative time interval would have to be returned. In this case, $\eta_i(\Delta t)$ is defined to return 0.

To analyze the timing behavior of a task set, the inverse of the density function, the so-called interval function $\delta_i(n)$, is also needed. The interval function returns the minimum time interval in which n events that activate τ_i may occur. The interval function is not part of the task description tuple, as it can be directly deduced from the density function.

Depending on the scheduling algorithm, a task is either assigned a fixed priority at system design time, or the priority of each task is calculated dynamically at runtime. An additional task, the so-called *scheduler*, decides on which task is to be executed next. This work focuses on *preemptive* systems in the following. In these systems, the scheduler can interrupt, or preempt, the running task and execute another task. For the scheduling analysis, the scheduling task itself may be modeled as the highest priority task in the system.

3.3 WCC: The WCET-aware C Compiler Framework

The WCET-aware C Compiler WCC [11] is a framework that specifically aims at optimizing the worst-case runtime behavior of a system. The compiler offers support for the Infineon TriCore TC1796 and TC1797 as well as for the ARM7TDMI architectures.

Our multi-task model is based on Gresser's well-known event model [13]. It allows for the description and analysis of arbitrarily triggered tasks with arbitrary deadlines. Using this model, arbitrary C functions can be annotated as so-called "entrypoints" in the C source code. An entrypoint marks the start of a hard real-time task. Further annotations can be used to specify the activation pattern as well as any deadline or priority in case of a fixed-priority scheduling algorithm. Based on this model, Luppold et al. [28, 29] integrated a schedulability-aware optimization framework based on ILP. Due to the relevance for the proposed multi-task and multi-core aware optimization framework, the mathematical ideas behind this are discussed in detail in Section 6.

Figure 2 depicts the internal structure of WCC. The solid arrows depict the design flow that can also be found in common compilers like, e.g., gcc: C Code is parsed into a high-level intermediate representation (ICD-C in WCC), which is then transformed by a code selector into an assembly-like low-level intermediate representation (the ICD-LLIR). Independent from any WCET optimization,

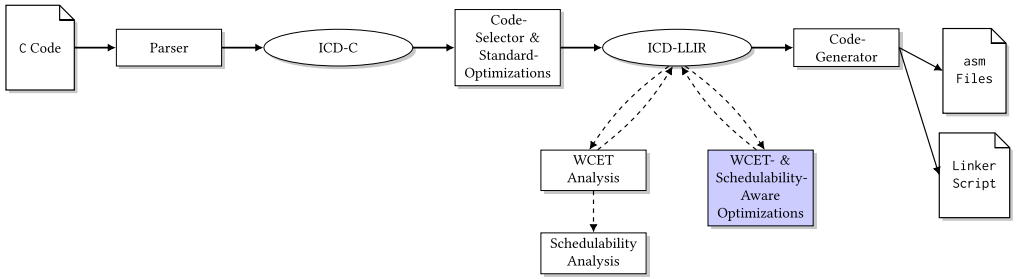

Fig. 2. Structure of WCC (based on Reference [11], updated). The solid lines show the classical flow of an optimizing compiler from C source code to assembly code plus linker script. The dashed lines denote WCC-specific extensions relevant for this article. The optimizations proposed in this article are performed in the highlighted box.

standard code optimizations like, e.g., loop unrolling or constant propagation and folding, are applied at both high- and low-level intermediate representations.

Once the C Code has been translated into the ICD-LLIR representation, both WCET and schedulability optimizations and analyses can be performed on this hardware-dependent code representation. For WCET analysis, an internal analyzer is available for the ARM7TDMI architecture. Additionally, WCC is tightly coupled to AbsInt aiT WCET analyzer [1] as an external analysis tool. The results of the WCET analysis are annotated back to both LLIR and ICD-C, allowing a precise knowledge about the worst-case timing behavior of each basic block within the code.

Based on the results of the WCET analysis, a subsequent schedulability analysis reveals whether the system will provably be schedulable in its current state. If it is, then no further WCET-oriented optimizations are necessary. If, however, the system is not schedulable, our schedulability-aware optimization framework may be called. It is able to account for interference delays due to preemptions for both fixed- and dynamic-priority scheduling. Furthermore, the additional overheads of the scheduler as well as context switching costs and timing penalties due to accesses to a shared memory bus are also taken into account. The framework will try to optimize the code running on each core to such an extent that all tasks on each core will afterwards be schedulable. Depending on the concrete optimization, WCET analysis and code modifications due to an optimization may be an iterative process.

In any case, after finishing any WCET- or schedulability-aware optimization, both WCET and schedulability analysis are performed once again as a safety measure to verify that the optimization did in fact establish schedulability on all processor cores.

4 ILP-BASED WCET OPTIMIZATION

This section gives a brief introduction into the basic ILP model used for the optimization framework. Section 4.1 gives a brief overview over commonly used symbols and notational conventions. Section 4.2 describes the general ILP constraints needed to model the WCET of each task for WCET minimizations. This model was originally proposed by Suhendra et al. [41] and then extended by Falk et al. [10] to perform a static instruction SPM allocation.

4.1 Notational Conventions and Basic Requirements

In any formulas, uppercase letters describe values that are calculated outside the upcoming ILP model or physical constants. Lowercase letters denote values that are added to the ILP as variables and are calculated by the ILP solver. For the sake of convenience, Table 1 shows the most frequently used symbols with a short explanation.

Table 1. Nomenclature Used within This Article

Abbreviation	Description
B_i	A basic block.
i	Index variable.
c_i	WCET of task τ_i.
$C_{i,Flash}$	WCET of one execution of the single basic block B_i if it is assigned to Flash.
$C_{i,SPM}$	WCET of one execution of the single basic block B_i if it is assigned to the SPM.
D_i	Deadline of task τ_i.
Δt	A time interval.
ΔT_B	The maximum busy window.
$\delta_i\,(n)$	Interval function of task τ_i. n denotes the number of events.
$\eta_i\,(\Delta t)$	Event density function of task τ_i.
e_i	Maximum preemption penalty inflicted to task τ_i.
$e_{i,j}$	Maximum preemption penalty inflicted to task τ_i by τ_j.
H	Hyper-Period of a task-set.
J_i	Activation jitter of task τ_i.
K	Number of times the currently analyzed task is executed within the analyzed time interval.
P_i	Period of task τ_i.
Q_{Flash}	Additional cycles due to jump correction code when jumping from Flash to SPM.
Q_{SPM}	Additional cycles due to jump correction code when jumping from SPM to Flash.
S_i	Net size of basic block B_i.
S_{SPM}	Overall size of the SPM.
τ_i	Task with index i.
u	Overall load of the system.
w_i	Execution time from the beginning of basic block B_i until the end of the block's function.
x_i	Indicator variable denoting whether basic block B_i is assigned to the SPM ($x_i \equiv 1$) or not ($x_i \equiv 0$).

4.2 Static Single-task Instruction SPM Allocation

The static single-task instruction SPM allocation model operates on a task's Control Flow Graph (CFG), as illustrated using the exemplary CFG from Figure 3(a). The goal of the optimization is to decide which basic blocks of a given task should be statically assigned to the fast-but-small SPM and which blocks should remain in main memory. Moving any basic block into a different memory may change the execution path that leads to the WCET. Additionally, after moving a basic block, additional instructions may need to be added to regain the original control flow on an assembly level [10]. As a result, the optimization problem can be seen as a version of the knapsack problem, where both weight and value of each item may change during the optimization.

The proposed ILP-based solution for this problem proceeds as follows: In two subsequent analyses, the WCET for each basic block is calculated twice: once with all blocks in slow Flash memory, and once with all basic blocks being allocated to the SPM. For this second run, the SPM is virtually increased such that the task fits completely into SPM for the analysis. The timing gain G_A for one single execution of basic block A is calculated by simply subtracting its execution time in SPM $C_{A,SPM}$ from the execution time if located in Flash $C_{A,Flash}$. To avoid loops in the ILP model, loops in the CFG are converted into a meta-block (depicted in Figure 3(b)). A variable w_L is introduced that bounds the maximum execution time one single iteration of a given loop. Then, another variable w_{Loop} is introduced that embeds w_L into the ILP model. w_{Loop} bounds the accumulated execution

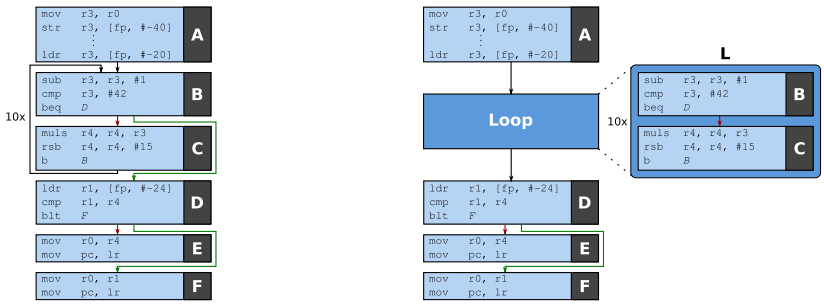

Fig. 3. Sample control flow graph of a task (a) and its representation with the loop substituted using a meta block (b).

time of the CFG starting at the entry block of the loop, covering all possible iterations of the loop and its successors. As a maximum loop bound of 10 is used in the example in Figure 3(a), w_L is multiplied by a factor of 10.

Equations (2) to (9) show the resulting ILP model:

$$w_A \geq C_{A,Flash} - x_A \cdot G_A + w_{Loop}, \tag{2}$$
$$w_{Loop} \geq C_{B,Flash} - x_B \cdot G_B + 10 \cdot w_L + w_D, \tag{3}$$
$$w_L \geq C_{B,Flash} - x_B \cdot G_B + w_C, \tag{4}$$
$$w_C \geq C_{C,Flash} - x_C \cdot G_C, \tag{5}$$
$$w_D \geq C_{D,Flash} - x_D \cdot G_D + w_E, \tag{6}$$
$$w_D \geq C_{D,Flash} - x_D \cdot G_D + w_F, \tag{7}$$
$$w_E \geq C_{E,Flash} - x_E \cdot G_E, \tag{8}$$
$$w_F \geq C_{F,Flash} - x_F \cdot G_F. \tag{9}$$

The x_i variables are binary ILP variables that, if set to 1 by the ILP solver, define that the basic block will be moved to the SPM. Otherwise, it will be allocated to the shared Flash memory. As can be seen, each basic block is associated with an ILP integer variable w, which denotes the accumulated WCET of that basic block including all possible paths through succeeding blocks. As a result, the task's entry block w_A holds a safe over-approximation for the WCET of the task.

Additional constraints limit the number of basic blocks that may be moved to the SPM without overflowing the physically available memory.

In case one basic block is placed in the SPM while its successor resides in the Flash memory (or vice versa), potential additional jump penalties have to be considered. This stems from the fact that, e.g., a previous relative jump instruction has to be replaced by an indirect one (to cope with the physical address offset of the two memory regions). If, e.g., basic blocks B and C from the CFG in Figure 3(a) are placed in the SPM, while all others reside in the Flash memory, two jumps have to be fixed potentially: As there was no explicit jump instruction from basic block A to B, a jump has to be inserted here. Additionally, the physical memory addresses of the SPM and the Flash may differ greatly, therefore exceeding the maximum offset of the relative jump instruction used from B to D. Thus, the relative jump instruction from B to D has to be replaced with a more costly indirect jump, for which the target address has to be loaded into a register previously. We

previously elaborated on this extensively in Reference [36]. Due to the missing novelty, we do not explicitly state these constraints here.

Due to the fact that this optimization only targets single-task systems, the goal is to minimize *the* WCET of the system. Therefore, the ILP's objective may simply be set to minimize w_A:

$$\min w_A. \tag{10}$$

For multi-tasking setups, the constraints provided in this section can be repeated for all tasks on each core. As a result, for each task τ_i, there will be a corresponding integer variable w_i in the ILP that holds a safe over-approximation of one single execution of this task, and therefore its WCET. To keep a common notation, we use a variable c_i for this WCET of task τ_i.

5 MULTI-CORE ILP EXTENSION

In a multi-core system, any access to the shared memory by a task τ_i running on one given core depends on the current state of the memory bus and the bus arbitration scheme. Section 5.1 describes the necessary extensions to the previously introduced ILP model to provide a safe estimate on a task's WCET c_i in a multi-core system.

As we introduce some assumptions about the bus architecture used in Section 5.1, we discuss their effects on the average-case execution times in Section 5.2 to verify their usability in practice.

If multiple tasks are running on one given core, then a given task τ_i may be preempted by any higher-priority task τ_j. In this case, τ_i may suffer from additional timing delays due to the context switching costs. Analogous to the WCET c_i, these preemption costs are highly bus-dependent. Section 5.3 therefore proceeds by introducing an ILP variable $e_{i,j}$, which safely bounds the preemption penalty inflicted on a task τ_i by one eviction by another task τ_j running on the same core.

5.1 Bus-aware WCET Calculation

As the timing of any shared memory access depends on the current state of the bus and its arbitration policy, simply applying single-task single-core methods (cf. Section 4) may lead to suboptimal results (as bus effects are ignored). The base ILP model from Section 4 derives its timing gains per basic block by two analysis runs, once with all basic blocks placed in the SPM and once with all blocks located in Flash. This inherently assumes that the time required to execute a basic block does not depend on the current point in time. Yet, as the state of the shared bus typically depends on the current point in time, the timing gains per basic block do so as well.

Example: We consider the basic block E from the sample control flow graph from Section 4.2 with a derived timing gain of G_E. The timing gain was calculated by subtracting the execution time of basic block E when being placed inside the private SPM from the execution time when placed inside the shared Flash memory. Note that for both timing analysis runs used for deriving the timing gains per basic block, *all* basic blocks were placed either into the Flash or into the SPM. If any predecessor of basic block E is being placed inside the private SPM (hence, having a shorter execution time), yet E remains in the shared Flash memory, then the bus state during the execution of basic block E is potentially different from the one analyzed during the "all-in-Flash-analysis." This in turn potentially invalidates the precision of G_E.

To cope with this property of a multi-core architecture without increasing the complexity of the ILP formulation severely, we make the following assumptions on the used bus architecture: The shared bus is arbitrated using a time division multiple access (TDMA) policy with fixed slot sizes. A TDMA policy guarantees safe upper timing bounds for each shared memory access due to its fixed slots. Additionally, it allows us to optimize each core individually, as the bus access behavior of one core cannot interfere with another. Yet, the execution time of a basic block residing in the shared Flash memory is still dependent on the bus state when being executed. We reduce

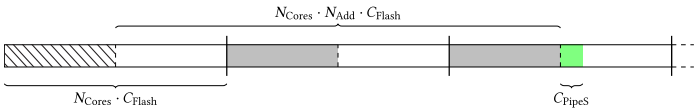

Fig. 4. Exemplary TDMA Schedule for a jump correction from shared flash to private SPM for $N_{\text{Cores}} = 2$ and $N_{\text{Add}} = 2$.

the timing dependency of basic blocks executed from the shared Flash memory to a minimum by assuming slot lengths to be equally sized and fixed to the latency of one Flash memory access. With each Flash memory access, exactly one instruction is fetched. Therefore, only one access can be executed per core in its corresponding TDMA slot. Moreover, if a core *does* initiate an access to the shared Flash memory, we now precisely know the TDMA offset after this access finished (as the request can only be granted at exactly one cycle in the TDMA schedule). Therefore, all timings of basic blocks residing in the shared Flash memory remain the same, independent from the timings of the preceding basic blocks, as the fetching of the first instruction acts as a "synchronization point." Due to the jump corrections, the number of instructions in a basic block may increase (cf. Section 4.2). Therefore, variable timings still exist when performing a jump from the private SPM to the shared Flash memory or vice versa.

We previously showed that the actual timing penalty of such a jump is dependent on the source and target memory [36]. In our multi-core setup, it is not only dependent on the memory timings themselves, but also on the current TDMA offset (current instant of time in regard to the TDMA schedule) when the jump should be executed. We previously presented an instruction SPM allocation optimization [37] targeting multi-core single-task architectures that dynamically predicts all potential TDMA offsets according to the allocation. Extending this precise optimization to a multi-core multi-task system turned out to be computationally infeasible due to its significant complexity increase. To take the shared bus into account without increasing the complexity of the model significantly by calculating all potential TDMA offsets, we introduce a jump penalty value that is dependent on the memory timings, the overall bus schedule, and the number of instructions to be inserted. If a jump from basic block A to basic block B has to be repaired, then we will refer to A as the *source* basic block and B as the *target* basic block. A jump has to be repaired if there was no explicit jump instruction previously (but now required) or a previous jump instruction's distance is not sufficient anymore.

Jump from shared Flash to private SPM: The additional instructions will be inserted in code residing in the Flash memory. Since we restricted the slot lengths, the TDMA offset at the execution of the jump correcting instructions is known. The fetching of each instruction can only happen at the very first cycle of the core's TDMA slot (all instructions of the source basic block reside in the shared Flash memory). Hence, there is no uncertainty of the TDMA offset at this point. The additional timing penalty can be calculated as follows:

$$Q_{\text{Flash}} = N_{\text{Add}} \cdot N_{\text{Cores}} \cdot C_{\text{Flash}} + C_{\text{PipeS}}, \tag{11}$$

where N_{Add} is the additional number of instructions that have to be inserted, N_{Cores} is the number of cores, and C_{PipeS} is the number of additional cycles to refill the pipeline from the SPM. C_{Flash} denotes the access latency of the Flash memory.

The basic principle of Equation (11) is depicted in Figure 4 exemplary for $N_{\text{Cores}} = 2$ and $N_{\text{Add}} = 2$. The thick vertical bars depict the end of a full TDMA schedule period, whereas the dashed vertical bars are delimiter symbols for a core's slot inside a TDMA period. We assume that the program is allocated to the core that owns the first slot of every TDMA period. The first hatched slot marks the execution of the last instruction before the additional jump correcting instructions. It

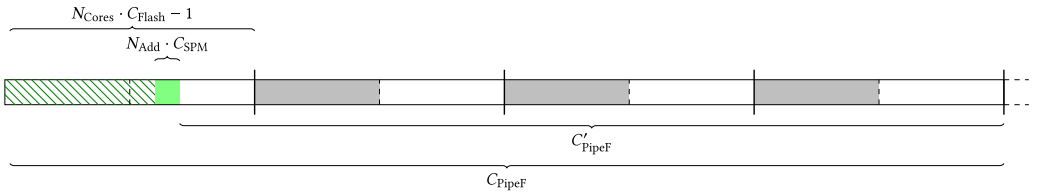

Fig. 5. Exemplary TDMA schedule for a jump correction from private SPM to shared flash for a processor with a 3 stage pipeline and $N_{\text{Cores}} = 2$.

takes $4 \cdot C_{\text{Flash}}$ cycles to execute two additional instructions that reside in the shared Flash memory (*reminder:* each slot length is set to C_{Flash}). When executing the actual jump into the private SPM, the processor has to refill its pipeline with instructions from the SPM. As this memory is private, the processor is not required to access the bus and therefore takes a constant amount of cycles (C_{PipeS}).

Jump from private SPM to shared Flash: The additional jump instructions are placed inside the SPM. The additional timing costs due to these instructions are assumed as follows:

$$Q_{\text{SPM}} = N_{\text{Add}} \cdot C_{\text{SPM}} + C_{\text{PipeF}}, \tag{12}$$

where C_{SPM} denotes the access latency of the SPM. C_{PipeF} represents an overapproximation of additional cycles required to refill the pipeline from the Flash memory. It is partially dependent on the current bus offset, since we have to access the shared Flash memory at an unknown bus offset.

The basic principle of Equation (12) is depicted in Figure 5 exemplary for a processor with a three-stage pipeline. The hatched area depicts the execution of instructions inside the private SPM prior to the additional jump-correcting instructions, whereas the succeeding solid area depicts the execution of the additional instructions. When executing the actual jump into the the shared Flash memory, the processor has to refill its pipeline again with instructions residing in the Flash. As only a single fetch from the Flash is allowed per TDMA period, the processor requires least three TDMA periods to refill its three-stage pipeline. C'_{PipeF} denotes the precise amount of cycles required for this refill at this particular point in time. As it can be seen in the diagram, C'_{PipeF} is dependent on the TDMA offset at which the jump from the private SPM to the shared Flash memory is executed. C_{PipeF} is a safe over-approximation that is used to avoid the requirement to predict all possible TDMA offsets for a jump. The maximum pessimism is $N_{\text{Cores}} \cdot C_{\text{Flash}} - 1$, since the subsequent accesses happen again at a known offset.

These additional penalties are added to all basic blocks with successors. For the purpose of illustration, we use ILP inequation (6) to demonstrate this:

$$\begin{aligned} w_D \geq\ & C_{D,\mathit{Flash}} - x_D \cdot G_D + w_E \\ & + Q_{\text{Flash}} \cdot (\overline{x_D} \wedge x_E) + Q_{\text{SPM}} \cdot (x_D \wedge \overline{x_E}). \end{aligned} \tag{13}$$

In case basic block D resides in the Flash memory and its successor E in the SPM, the term $\overline{x_D} \wedge x_E$ evaluates to 1, hence the timing penalty for a jump from the shared Flash to the private SPM Q_{Flash} will be added. For the opposite case, x_D is set to 1 and x_E to 0, thus the timing penalty for a jump from SPM to shared Flash Q_{SPM} memory will be added. The logical $\wedge$ operator can be expressed as a set of simple ILP inequations as shown in Reference [31].

5.2 Average-case Impacts of Architecture Assumptions

The rather strict appearing requirement on the TDMA schedule to minimize the uncertainty of a shared memory access raises the question of its usefulness in practice. We therefore examined

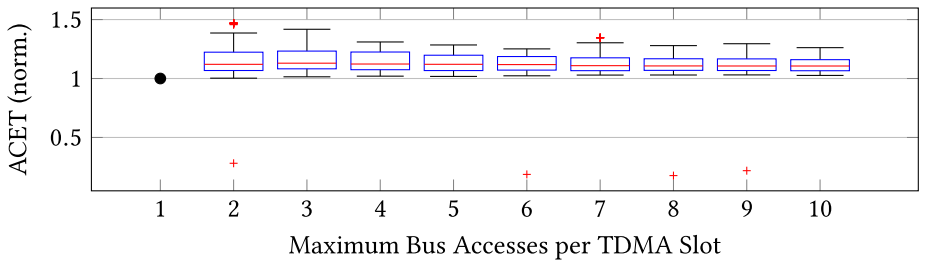

Fig. 6. Normalized average-case execution times for varying numbers of accesses per TDMA slot. ACETs were generated with the cycle-true instruction set simulator CoMET [42] applied to the ARM7TDMI multi-core model described in Section 3.1.

the correlation of average-case execution times and maximum number of shared memory accesses inside a TDMA-scheduled architecture. The results of this case-study are depicted in Figure 6.

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× 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- [14], UTDSP- [6], DSPStone- [48], MediaBench- [22], MiBench- [15], NetBench- [32], PolyBench- [27], and StreamIT-benchmark suites [40]. 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 [42] and an existing ARM7TDMI multi-core model [19] that resembles the system described in Section 3.1. No specific SPM allocation was done, hence all instructions reside in the shared Flash memory.

Example: A benchmark took on average (median) 1.12× 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 6 shows that besides four outliers, none of the evaluated benchmarks show a degraded ACET when using a minimum TDMA slot length (one 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 not typically degrade the timing of a system and is therefore a valid constraint for our optimization.

5.3 Calculation of Preemption Penalties

If a task τ_i is preempted by a higher-priority task τ_j, then the processing unit's pipeline must be refilled when τ_i resumes. For the ARM7TDMI architecture with its three-stage pipeline that is used through this article, this means that two additional accesses to memory must be made by τ_i after each preemption. On our architecture, an access to the local SPM memory takes place within one CPU cycle. The maximum access time to the shared memory stems from unknown TDMA offsets after resuming from the preempting task. Thus, the additional delay of task τ_i due to one preemption by task τ_j is:

$$e_{i,j} = \begin{cases} 2 & \text{if whole task } \tau_i \text{ is in SPM.} \\ 2 \cdot P_{Bus} & \text{else.} \end{cases} \tag{14}$$

P_{Bus} is the maximum penalty a task has to wait until it gets bus access. Since we consider TDMA with identical slot lengths for all cores, the penalty is equal for each core. As previously shown by

Kelter [19] for the TDMA bus, P_{Bus} is safely upper-bounded by the sum of the maximum memory access time plus the time of one period of the TDMA schedule minus one time unit.

If all basic blocks of task τ_i are located in the private SPM, then a preempting task can obviously not create any preemption penalty due to varying TDMA offset, as no basic block of this task resides in the shared Flash memory. Due to the fact that we consider arbitrary asynchronous task systems, we do not know the time at which a task is preempted by another task. Therefore, we must account for the global worst-case scenario, which is that we have to account for the worst-case preemption penalty as soon as at least one block of τ_i is located in non-local memory.

The "case" structure of Equation (14) can easily be expressed as a conditional constraint in the ILP that sets $e_{i,j}$ to the pre-computed value P_{Bus} if the sum over all basic block SPM indicator variables x_b for τ_i is greater than 0. The exact formulation is straightforward and can be found, e.g., in Bisschop [5].

6 ILP-BASED SCHEDULABILITY OPTIMIZATION

The previous Section 5.1 showed how the WCET c_i of each task τ_i can be modeled while respecting bus effects. Additionally, Section 5.3 described how to model the ILP variable $e_{i,j}$ that bounds the maximum additional delay inflicted on task τ_i by one preemption by a higher-priority task τ_j.

This section proceeds by showing how these variables can be combined with each task's timing properties to provide a schedulability-aware optimization framework. The ILP constraints will ensure that *any* solution that is returned by the ILP solver will lead to a schedulable system. If the task set cannot be optimized to this extent, then the resulting ILP will be infeasible.

The constraints that have to be added to the ILP depend on the scheduling algorithm under which the task set is scheduled. Therefore, Section 6.1 and Section 6.2 give an overview of the ILP constraints necessary to model the schedulability constraints for EDF and fixed-priority scheduling, respectively. The approach on EDF and fixed-priority systems was previously presented in Reference [29] for single-core systems. For fixed-priority scheduling, the ILP that was presented in Reference [29] grows quadratically with the number of task activations that have to be modeled. In contrast to this, the approach presented in the following improves on the previously presented approach by growing only linearly with respect to the number of task activations. At the same time, the new approach does not suffer from any additional limitations over the previous one. Both EDF and fixed-priority scheduling allow for modeling of arbitrarily triggered tasks with arbitrary deadlines.

Due to the fact that both approaches are solely relying on ILP constraints, they do not require any concrete optimization objective. This allows the system designer to set an arbitrary optimization goal like, e.g., minimizing the overall utilization. If no optimization goal except from schedulability exists, then a dummy objective may be set, resulting in the ILP solver returning any valid solution. Due to the fact that we assume that the allocation of a task to a corresponding core is fixed, the ILP can either be solved for all tasks on all cores at once or for each core separately.

6.1 Earliest Deadline First Scheduling

Schedulability constraints including task preemption penalties were previously presented in Reference [29]. Based on Baruah [2], schedulability can be modeled by enforcing that the amount of required computation time is less than or equal to the amount of available computation time:

$$\Delta t \geq \sum_{\forall \tau_i} \left[\eta_i \left(\Delta t - D_i \right) \cdot \left(c_i + e_i \right) \right], \tag{15}$$

where Δt denotes one time interval that is to be analyzed. $\eta_i \left(\Delta t - D_i \right)$ returns the number of activations of task τ_i that may occur within $\Delta t - D_i$, and must thus have finished within Δt.

This is done for all tasks, and the number of activations of each task is multiplied with its respective maximum computational demand (i.e., its WCET plus additional preemption penalties). If the sum of the computational demand over all tasks within Δt exceeds Δt, then at least one task will not be able to finish within its deadline and the system is not schedulable. Otherwise, if for all Δt the inequation holds, then all tasks will provably always be able to finish without violating any timing constraints.

Due to the nature of a dynamic scheduling algorithm, any task may have a higher priority than any other task at some point during execution. Therefore, it is hard to predict which tasks may evict a given task τ_i within the currently analyzed interval. As a result, the ILP integer variable e_i is introduced, which denotes the maximum penalty inflicted on τ_i by a preemption by any other task:

$$e_i = \max(\{e_{i,j}\}), \forall \tau_j, j \neq i. \tag{16}$$

The schedulability test has to be performed for all possible time intervals Δt, starting at an interval of 0 up to the task set's so-called maximum busy window ΔT_B. The maximum busy window is the maximum allowed time interval in which the system may be constantly busy without any idle phase. For periodic systems, the busy window is the task set's hyper-period, i.e., the least common multiple over all fundamental activation periods of all tasks. For tasks with non-periodic bursts, the task's timing behavior can be divided in an aperiodic and a periodic part. Therefore, the maximum busy window is the hyper-period of all periodic parts plus the maximum time interval in which aperiodic events may happen. If a task's activation pattern never does evolve into a recurring pattern, then the behavior of the task set will never exactly repeat. Thus, the hyper-period would become infinite and the task set's schedulability cannot be analyzed.

Due to the nature of interval-based timing analysis, the tightest sequence of events must happen at the small time intervals—independently from when it will occur in wall clock time; e.g., if $\eta_1(3) = 2$, then within *any* time interval of 3 time units starting at any moment in wall clock time, there must not be more than 2 activations of task τ_1.

Still, even when considering a discrete time space (e.g., CPU clock cycles as fundamental time unit), checking each and every point in ΔT_B is practically impossible. Fortunately, for EDF, preemptions can only occur if a new task is ready for execution. Thus, the test only has to be performed at the points of discontinuity in the event density function of all tasks. Due to the fact that Equation (15) is purely linear, the constraints can directly be added as ILP inequations.

Finally, it must be ensured that the system's overall load due to the periodically repeating task activations is not beyond 100%. Otherwise, situations might occur where all deadlines are met within the hyper-period, but some instances of the task set back up and finally, after multiple hyper-periods at a load beyond 100%, deadline misses will finally occur. The maximum valid system load can be enforced by adding the following constraint to the ILP:

$$H \geq \sum_{\forall \tau_i} \left[\eta_i^{\text{per}}(\Delta T_B) \cdot (c_i + e_i) \right], \tag{17}$$

with $\eta_i^{\text{per}}(\Delta t)$ being the event density function of task τ_i only including periodically recurring activation patterns. H is denoting the hyper-period of the system.

Due to the nature of being a constraint-based model of schedulability, any feasible solution to the ILP will provide an SPM allocation leading to a schedulable system. Additionally, due to the optimality of ILP, if any SPM allocation scheme exists that leads to a schedulable system, the ILP solver will find it. If no such solution exists, then the ILP is infeasible and the solver will return an error.

6.2 Fixed-priority Scheduling

The general idea behind optimizing systems scheduled under a fixed-priority algorithm is based on our previously presented approach [29]. However, this previous ILP model grows quadratically with the number of events that have to be modeled within the hyper-period of the task set. The approach presented in this section tackles these shortcomings by modifying the resource demand test for EDF scheduling such that it can be applied to fixed-priority scheduling. Subsequently, the size of the ILP model will only grow linearly with the number of events that have to be analyzed.

Since arbitrary deadlines can be modeled, multiple instances of the same task τ_i may be ready for execution at a given point in time. Therefore, K subsequent instances for each task τ_i are tested for their schedulability. The maximum number of instances K of a task τ_i, $\hat{K}$, equals the number of instances that are triggered within the system's maximum busy window: $\hat{K} \coloneqq \eta_i(\Delta T_B)$.

With this, the maximum computational time $v_{i,K,\Delta t}$ needed to finish the execution of K consecutive instances of task τ_i in a given time interval Δt can be expressed as:

$$v_{i,K,\Delta t} = K \cdot c_i + \sum_{j=0}^{i-1} \left\{ \eta_j(\Delta t) \cdot c_j + \sum_{n=j}^{i} \min[\eta_n(\Delta t) \cdot \eta_j(D_n), \eta_j(\Delta t)] \cdot e_{n,j} \right\}. \tag{18}$$

The first term accounts for the maximum execution time needed by K instances of τ_i without any penalties due to preemptions by higher-priority tasks. The term $\eta_j(\Delta t) \cdot c_j$ in the outer sum models the WCETs of all tasks with a priority j higher than τ_i (i.e., $j < i$). Finally, the inner sum stands for additional preemption penalties due to context switches. $e_{n,j}$ denotes the maximum costs of *one* preemption of task τ_n by τ_j. The min [] term bounds the maximum number of times τ_j may preempt τ_n: The number of preemptions is trivially bounded by the number of instances of middle-priority tasks multiplied by the number of activations of the higher-priority tasks. However, this may not occur more often than the higher-priority task itself is scheduled for execution. Thus, the min [] term selects the smaller of both values. The min [] term solely depends on the currently analyzed time interval Δt and the constant deadlines D_n. Thus, it can be calculated off-line outside the ILP.

Two more constraints test whether the resource demand by τ_i is within its respective deadline:

$$o_{i,K,\Delta t} \equiv 1 \Rightarrow v_{i,K,\Delta t} \leq \Delta t, \tag{19}$$

$$o_{i,K,\Delta t} \equiv 1 \Rightarrow v_{i,K,\Delta t} \leq \delta_i(K) + D_i, \tag{20}$$

where $o_{i,K,\Delta t}$ is a binary decision variable that expresses if τ_i is schedulable within a given Δt. The $\Rightarrow$ notation denotes that iff $o_{i,K,\Delta t}$ equals 1, then the inequation on the right side of the $\Rightarrow$ *must* hold, too. These so-called conditionally enabled constraints may also be modeled directly in modern ILP solvers or be expressed as a set of regular inequations as proposed by, e.g., Bisschop [5]. The inequation term in Equation (19) models that the computational demand of τ_i within Δt is smaller than Δt; i.e., all K subsequent instances of τ_i finish within the given Δt. The inequation in Equation (20) is true if the computational demand is below the deadline of the Kth instance of τ_i. Due to the $\Rightarrow$ formulation, $o_{i,K,\Delta t}$ may only be set to 1 by the ILP solver if both inequations hold.

Since all tasks' WCETs are unknown prior to our ILP-based optimization, it cannot be stated *a priori* for *which* Δt a task will be schedulable (or if at all). However, the Kth instance of τ_i must finish within $\Delta t = \delta_i(K) + D_i$ the latest. Additionally, the Kth instance of τ_i cannot finish sooner than $K \cdot c_i$. For each instance K up to $\hat{K}$, this test is added for all Δt between $K \cdot c_i$ up to $\delta_i(K) + D_i$.

Thus, the complete ILP will contain constraints for *all* time intervals that may be valid. However, it is sufficient if, for one of these intervals, the resource demand will be lower than or equal to the

available computation time. This is easily modeled by another inequation for each K and task τ_i:

$$\sum_{\forall \Delta t} o_{i,K,\Delta t} \geq 1. \tag{21}$$

In analogy to EDF scheduling, it is necessary to constrain the system's load due to periodical activations not to exceed 100%. This again is achieved by Equation (17). Additionally, identical to EDF scheduling, the maximum busy window is bounded as well:

$$\Delta T_B \geq v_{i,\hat{K},\Delta T_B}. \tag{22}$$

7 GENETIC APPROACH

As an alternative to the ILP-based approach, this section introduces a genetic algorithm. It was originally proposed in Reference [37] for static SPM allocation of single-task multi-core systems. Although genetic algorithms cannot guarantee to return an optimal solution or even to find a valid solution at all, they have proven to be suitable for many types of large combinatorial problems in practice.

The following subsections give an overview of the approach to apply it to multi-core systems with multiple tasks assigned to each core. Compared to Reference [37], especially a new fitness function is proposed in the following.

7.1 Initial Population

The algorithmic representation of the instruction SPM assignment on a basic block level is straightforward: An arbitrary but fixed list of all basic blocks of a task set is created. Then, a new second list of same length consisting of binary variables is created. Each entry in this second list denotes whether the basic block at the corresponding position will be assigned to the SPM or not. This is done for each task set on each core individually. Since the task sets on different cores may contain different numbers of basic blocks, the lists will be of different size for each core's task set.

Finding a good initial population is crucial for a fast convergence of the genetic algorithm. As in traditional biological genetics, combining multiple very similar individuals will not tend to give major changes to the resulting individual. In this case, any real progress with regard to finding a close-to-optimal solution almost solely relies on the mutation process. To generate as diverse but still reasonable initial individuals as possible, this work uses the following approach:

- The first individual does not use any SPM at all, i.e., it is identical to the unoptimized program.
- For the second individual, all basic blocks on all cores are assigned to the SPM. Of course, this individual will probably be invalid, as the SPM is not large enough to hold the whole program (if it is, finding an optimum assignment is trivial). As a result, the repair function that is presented in Section 7.4 is used to repair this individual.
- For all other individuals, each block is assigned to the SPM with a probability of 0.5. As with the second individual, the repair function is used to make sure that each individual is valid.

7.2 Recombination

Due to the TDMA bus schedule that features fixed slot lengths for each core, the timing behavior of each core can be considered separately; i.e., one core's bus accesses are not delayed if basic blocks on another core are modified. Therefore, the recombination of two individuals A and B randomly selects one core to which a crossover will be applied. This prevents the situation that one core's schedulability will improve yet another core's schedulability will decrease within one

recombination. Thus, a fitness function can clearly indicate whether one individual outperforms another or not.

Once the core is chosen, a position in the list of basic block assignments is selected randomly. Recombination of two individuals on this core applies one-point crossover [12, p. 12] as proposed in Reference [37] for single-task multi-core setups. Then, all decision variables of individual A behind this randomly selected position in the list are exchanged by the respective decision variables of B. Tasks are not considered explicitly in this notion. This enables recombination to both remove basic blocks from the SPM for one task and to add basic blocks to the SPM of another task with one recombination.

7.3 Mutation

After recombination, one or multiple entries in the decision vector may be mutated randomly. To achieve this, two kinds of mutation were implemented: single-bit and multi-bit mutation.

For single-bit mutation, one entry in the decision vector is chosen randomly. This entry is then flipped with a probability of p, which can be defined by the user.

Multi-bit mutation mutates several entries of the solution vector of an individual in a two-stage process: Given a solution vector of length N, first, the maximum number M, $M \in [0, \ldots, N-1]$ of entries that are mutated is selected randomly. Second, a random entry E, $E \in [0, \ldots, N-1]$ is selected M consecutive times, and each time this entry E is flipped with a user-specified probability of p. As a result, solutions may be flipped up to N times in a solution vector of length N. Additionally, it is theoretically possible that by chance the same entry is chosen twice and also toggled twice, thus although mutations took place, the mutated individual is unchanged afterwards.

Experiments showed that multi-bit mutation reliably led to faster convergence towards a schedulable system. Therefore, we did not consider single-bit mutation in the upcoming evaluation.

7.4 Repair Function

For the instruction SPM allocation, an individual is invalid if more elements are assigned to the SPM than there is space available. A trivial solution would be to simply drop invalid individuals. However, an invalid solution may be near the optimum. Thus, a repair function is used to remove blocks from the SPM until the remaining basic blocks fit into the physically available memory.

To achieve this, in a first step, the solution vector is applied to the actual program. All blocks that are supposed to be moved to the SPM according to the solution vector are assigned to SPM without accounting for the needed space. Then, the correct control flow through the program on an assembly level is re-established by using the techniques described in Section 4.2 and Reference [36]. If the program fits into the physically available memory, then no corrections are performed and the repair function terminates.

Otherwise, the repair algorithm proceeds as follows: For each core, blocks in the SPM are randomly removed from SPM without re-applying control flow corrections until at most 80% of the SPM is still occupied on each core's SPM whose SPM region was previously overfull. In any case, the algorithm will always remove at least one block from an overfull SPM. Obviously, this may lead to a not optimally used SPM, as more basic blocks than necessary might be removed from SPM. For a more tight repair function, basic blocks would have to be removed randomly one-by-one, and control flow corrections would have to be re-run after each removal. However, running the control flow correction routines is quite runtime intensive and thus heavily decreases the genetic algorithm's performance—especially for larger programs.

A reduction to 100% (prior to running the control flow corrections) is not sufficient, as control flow correction may easily introduce a large amount of additional instructions. In practical

experiments, the somewhat arbitrary-looking 80% turned out to provide good results for the evaluated ARM7TDMI architecture. For different architectures, this parameter might have to be adapted.

Once the repair function finishes, control flow is corrected once again and it is tested whether the program adheres to memory size constraints. If not, e.g., due to a very unfavorable SPM allocation leading to extremely high jump correction costs, then the repair algorithm is repeated iteratively until the program actually fits into the SPM. Since each run of the repair algorithm removes at least one block from SPM, this approach is guaranteed to terminate with a valid individual.

7.5 Fitness Function

From a timing analysis view, a real-time system is either broken if at least one task might miss its deadline, or it is considered working if all tasks will provably meet their timing constraints. As a result, a naturally chosen (minimization) fitness function for the genetic algorithm is binary: 0 if the system is schedulable or 1 if it is considered broken.

Obviously, this measure is not suitable for a genetic algorithm, as it cannot distinguish which of two broken solutions is "better" or "worse." Due to lacking guidance of such a binary fitness function, the genetic algorithm degrades to a purely randomized search of the design space. A measure like minimizing the difference of each task's WCRT minus its deadline will not work either, as the WCRT of a task in the broken system might easily be infinite. Thus, we designed a sensitivity analysis to estimate "how broken" an individual is. The basic idea is to calculate how many tasks have to be removed from the system to achieve schedulability. The algorithm is based on the previous ILP optimization model from Section 6. Each task τ_i's runtime behavior is denoted by the ILP integer variable c_i. For the sensitivity analysis, this variable c_i is defined as follows:

$$c_i = b_i \cdot C_i, \tag{23}$$

where C_i is the constant WCET of task τ_i as returned by any WCET analyzer tool like AbsInt `aiT`. b_i is a binary decision variable whose value may be freely chosen by the ILP solver. These c_i variables are then used to describe a task's WCET in the schedulability constraints, as proposed in Section 6. The ILP solver can thus choose to remove τ_i from the task set by setting b_i to 0 to achieve a feasible ILP (and subsequently a schedulable task set).

The ILP's objective function finally ensures that only as few tasks as possible are removed:

$$\max \sum_{\forall \tau_i} b_i. \tag{24}$$

Using the constraints presented before, this approach works for both fixed-priority and EDF scheduling algorithms. Unfortunately, this measure is still very coarse-grained. For sets of N tasks, only N values are available for the fitness function. Therefore, the fitness function was divided into a primary and a secondary fitness measure:

Definition 1 (Relative fitness of two individuals). An individual i that describes a task set consisting of N tasks is considered to be fitter than another individual j if less tasks must be removed to achieve schedulability. If the numbers of removed tasks are identical, then i is fitter, iff

$$\sum_{i=0}^{N-1} C_i \cdot b_i < \sum_{j=0}^{N-1} C_j \cdot b_j, \tag{25}$$

where C_i and C_j are the WCETs of each task in each individual, respectively. b_i and b_j are 1 if the respective task may stay in the system and 0 if the task must be removed from the system to achieve schedulability.

This way, the genetic algorithm is now able to compare the fitness of different individuals. It must, however, be noted that, from a system designer's point of view, the fitness of a hard real-time system with regard to its timing actually *is* binary. It does not matter *how many* tasks may miss their deadline or by *how much* they miss it. Once one task may miss its deadline by even one CPU cycle, the system must be considered functionally broken. It is therefore utterly impossible to give any precise measure on the amount by which the system is broken. The only valid purpose of the measure presented here is to guide the genetic algorithm in the correct direction.

We also investigated different fitness measures like, e.g., an augmentation factor. The augmentation factor $F \in [0, 1]$ describes by how much to scale down all tasks' WCETs such that the system is schedulable; i.e., an individual with a large augmentation factor would be fitter than one with a smaller one. If the augmentation factor equals 1, then the task set on that core is schedulable.

Our experiments, however, showed that evaluation of the augmentation factor is only marginally faster than the ILP-based fitness measure. At the same time, using the augmentation factor as fitness function led to significantly slower convergence of the individuals. Both ILP-based and augmentation factor-based approaches could estimate the fitness of an individual in well under one second (with Gurobi as ILP solver). Compared to the fact that the WCET analysis of a task set will easily take several minutes, faster convergence easily outweighs any possible minor differences in runtime performance of the fitness evaluation. Thus, we stuck to the ILP-based fitness function and did not include the use of the augmentation factor in our evaluation.

8 EVALUATION

We used an ARM7TDMI multi-core setup with 1, 2, 4, and 8 cores, respectively, for our evaluation. Each core has a private SPM and can access the shared Flash memory via a TDMA-scheduled bus. All TDMA slots are equally sized and set to the latency of one Flash access, here assumed to be six cycles. One access to the private SPM takes one cycle. The size of the SPM is crucial for the evaluation to provide meaningful results. If the SPM size is too small, then it is impossible to achieve any meaningful improvements, as little to no basic blocks can be assigned to the fast memory. If the SPM is too big, then the optimization is trivial, as most or all timing-critical blocks can be moved to the SPM. As a trade-off, we chose the SPM size of each core to be 40% of the size of the respective task set running on that core.

Due to the lack of suitable multi-task benchmark sets, we created the task sets by randomly combining tasks from the Mälardalen benchmark suite [14]. We took the benchmarks annotated with flow facts from the TACLe benchmark suite [9]. An identical number of tasks was created per core. For each configuration of cores, we created systems with 2, 4, 6, and 8 tasks per core. To compensate for statistical outliers, 20 task sets were created for each configuration.

Compilation of the task sets was performed using the WCET-aware C Compiler framework (WCC) [11] with optimization level -O2. After applying these standard code optimizations, all systems were then analyzed using the bus-aware WCET analyzer by Kelter [19]. Then, UUniFast [3] was used to randomly create periods for each task in each task set. For each task set, periods were assembled such that the task set has an approximate load of $0.8, 1.0, \ldots, 2.2$ if it is allocated in Flash memory; i.e., for each task set, eight different system loads are evaluated.

The deadline of each task was uniform randomly chosen between 0.8 and 1.2× the task's period. Additionally, a jitter of up to 1% of each task's period was chosen uniform randomly.

This leads to 80 task sets for each of the four different core sizes, each with eight different system loads. Therefore, 2,560 configurations are evaluated for each optimization. The evaluation was performed using the schedulability-aware ILP and the genetic approach. Additionally, the ILP without schedulability constraints (cf. Section 4) that simply minimizes the sum over all WCETs

was evaluated as reference. These three optimizations were performed for both DMS and EDF scheduling. Thus, 15,360 evaluations were performed in total.

Due to the uniform distribution used by Bini and Butazzo and the large range of possible WCETs of the used benchmarks, the resulting hyper-period of the system may also become very large; e.g., the WCET of the `lms` benchmark is beyond 10M CPU cycles while, e.g., `minver`'s WCET is only roughly 70k cycles. For sets of eight tasks, the resulting hyper-period often exceeded 10^{20} cycles. This renders both schedulability analysis and the optimization very hard, as the schedulability test has to be performed up to this value. To circumvent this issue, we applied the solution proposed by Xu [45]. This way, the randomly generated periods are slightly tightened for the analysis and optimization, such that the least common multiple (and thus the hyper-period) can be decreased by multiple orders of magnitude.

Systems with a load below 1.0 may already not be schedulable due to multiple reasons: First, the deadline of a task may be as tight as 80% of its period, therefore, even with a load below 1.0, deadlines may easily be violated. Second, due to the selected jitter and the period adjustment described above, the load that appears in the optimization and analysis is always slightly higher than the originally aspired target load. Finally, DMS is not an optimal scheduling algorithm, and for both DMS and EDF, context-switching costs were neglected in the period calculations above but are, of course, accounted for in any analyses and optimizations.

For loads of 1.0 or higher, all systems are certainly not schedulable if no SPM allocation is performed. The evaluation subsequently focuses on repairing as many of these unschedulable systems as possible. We evaluated the optimization on Intel XEON compute servers with Gurobi as ILP solver. We restricted Gurobi to use at most four threads in parallel. For all benchmarks, a time cap of one hour per core to be optimized was set; i.e., for the two-core setup, there was a time cap of two hours, while each eight-core benchmark could run eight hours prior to being terminated. This time limit includes the basic compilation of the input C files as well as all WCET and schedulability analyses and the optimization itself. All results on repair rates shown below are based on separate WCET and WCRT analyses after finishing the proposed ILP and genetic optimizations.

Figures 7 to 10 show the results of the evaluation for the different numbers of cores. For each multi-core setup, there are four diagrams showing the results of systems with 2, 4, 6, and 8 tasks per core. The X-axis of each diagram denotes the original system load if the whole task set is in Flash memory. The Y-axis gives the percentage of task sets that could be repaired. A repair rate of 100% means that all 20 task sets that were evaluated with this original load could be repaired. The first four bars show the results when using DMS scheduling. The first bar in the diagram shows the percentage of systems that is already schedulable without any optimization. For a load of 1.0 or higher, this is obviously 0. Following, the second bar shows the results of our proposed schedulability-aware ILP framework, as proposed in Section 6. As a reference, the third bar shows the results when omitting any schedulability-aware constraints in the ILP but simply minimizing the sum over all WCETs of all tasks on each core, which corresponds to the ILP model described in Section 4. The fourth bar finally shows the results when applying the genetic algorithm from Section 7. The results are then repeated in the same order for EDF scheduling instead of DMS.

Apart from minor statistical spikes, the results' trend is consistent for all core and task set sizes: Even for systems with eight tasks and eight cores, thus running 64 tasks in total, the ILP-based approach is able to optimize a significant amount of systems up to an original load of 200%. This basically means that the computational demand could be cut in half by our optimization framework.

It can further be seen that EDF scheduling is sometimes performing worse than DMS. This stems from the fact that, as soon as preemption penalties are no longer neglected, EDF is no longer an

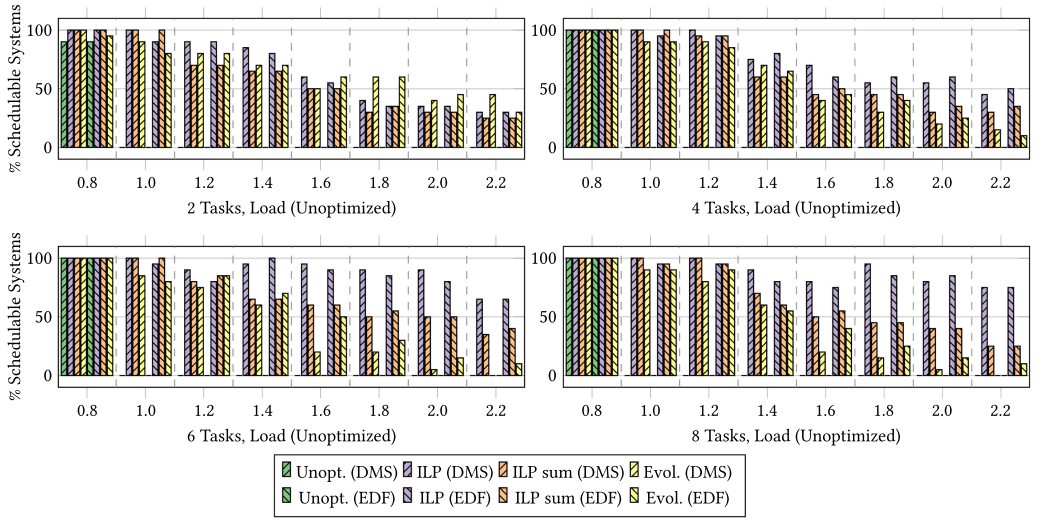

Fig. 7. 1-core system. The X-axis of each graph denotes the original load of randomly assembled task sets. The Y-axis denotes how many percent of these task sets are schedulable with the respective scheduling algorithm and optimization.

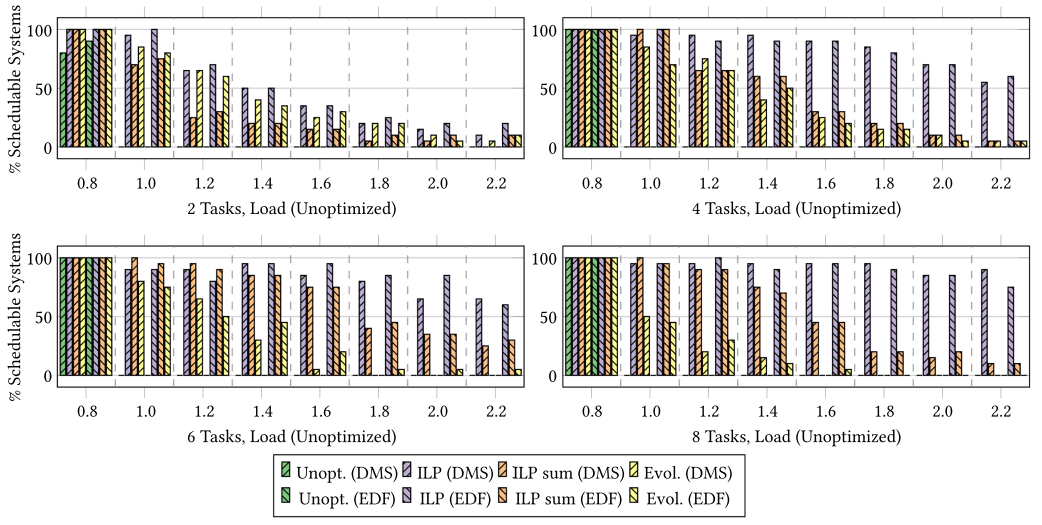

Fig. 8. 2-core system. The X-axis of each graph denotes the original load of randomly assembled task sets. The Y-axis denotes how many percent of these task sets are schedulable with the respective scheduling algorithm and optimization.

optimal scheduling algorithm. A more sophisticated analysis of this issue has previously been given by, e.g., Phavorin et al. [38].

For all numbers of cores, it can be observed that a higher percentage can be repaired for larger task sets than for small task sets. This at-first-glance counter-intuitive result stems from the fact that the SPM size was chosen relatively to the program size. Therefore, for a large task set, the *absolute* size of the SPM is larger than for the small task sets. This enables both ILP and genetic

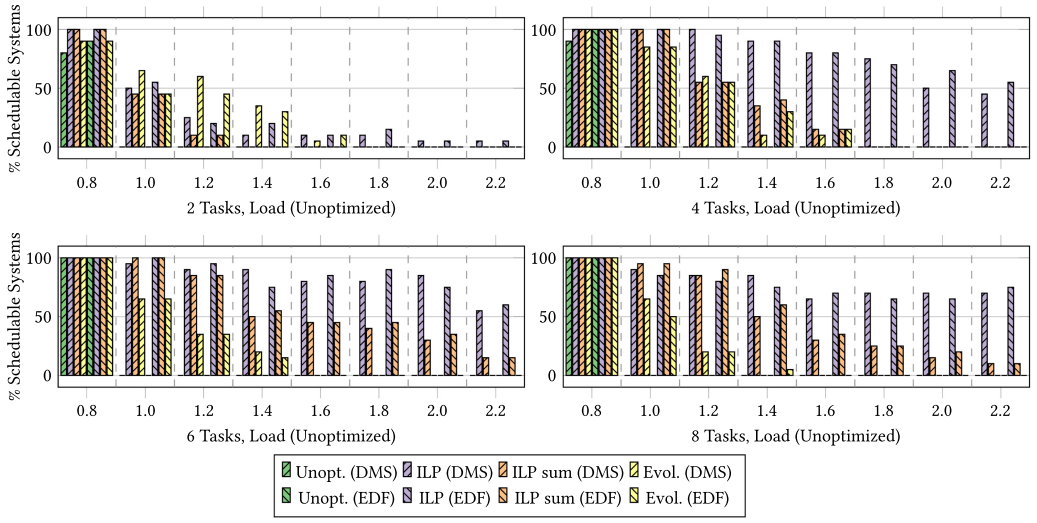

Fig. 9. 4-core system. The X-axis of each graph denotes the original load of randomly assembled task sets. The Y-axis denotes how many percent of these task sets are schedulable with the respective scheduling algorithm and optimization.

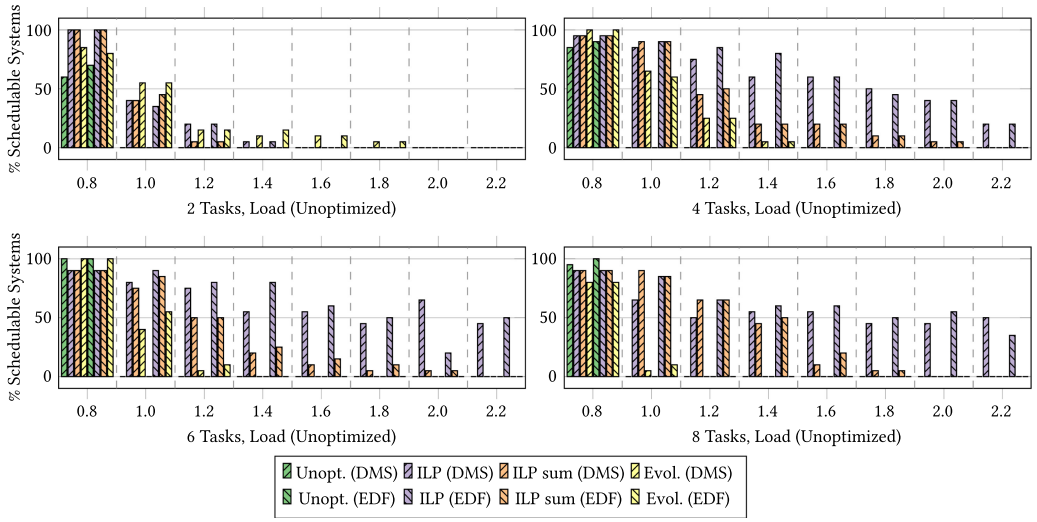

Fig. 10. 8-core system. The X-axis of each graph denotes the original load of randomly assembled task sets. The Y-axis denotes how many percent of these task sets are schedulable with the respective scheduling algorithm and optimization.

approaches to optimize these tasks more aggressively, which is critical for the system's schedulability.

For small task sets consisting of only two tasks, the genetic algorithm sometimes outperforms the ILP approach with regard to the number of repairable systems. This WCET-formulation in the underlying ILP model as proposed by Suhendra et al. [41] and Falk et al. [10] is more pessimistic than a full-fledged WCET analysis; e.g., our used ILP models cannot account for different WCETs of a function dependant on its calling context. The genetic algorithm that directly uses the results

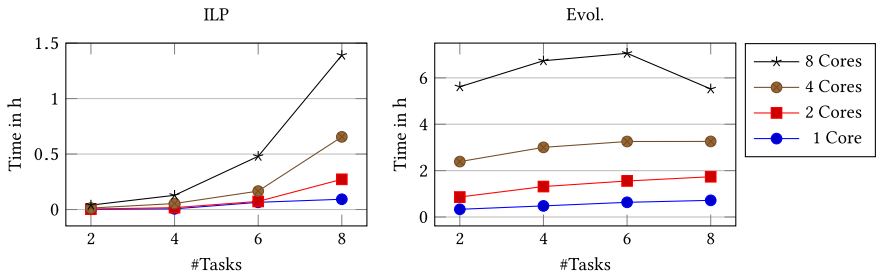

Fig. 11. Execution time of WCC for both ILP-based and evolutionary algorithm–based SPM allocation. The timings include all stages of the compilation and all necessary WCET analyses. The graphs depict the arithmetic means over all evaluated task sets for the denoted number of cores and tasks. Note the different scales on the Y-axes that were applied to provide better readability.

of the precise WCET analysis does not suffer from this pessimism. The ILP could be extended to handle such contexts on the drawback of further solving complexity. For larger task sets, it can be observed that the genetic algorithm fails to repair almost any system. This mainly stems from the fact that WCET analysis is very time-consuming. For the genetic algorithm, this analysis has to be performed on each individual as a prerequisite for the schedulability analysis and fitness evaluation. This drastically limits the number of individuals that can be evaluated before hitting the timeout.

Figure 11 shows the runtime of all evaluated task sets for both repairable and non-repairable systems including all WCET analyses. In other words, the graphs depict the time a system designer has to wait on average to find out whether the system could automatically be repaired or not. As we applied a timeout of 1 h per processing core; this means that, e.g., for a two-core system, an execution time of 2 h or 7,200 s is accounted for if a timeout is met. The diagram shows that the ILP-based approach performs significantly better with respect to the needed execution time than the genetic approach. This stems from the fact that the ILP-based approach needs massively less WCET analyses. In fact, prior to solving the ILP, two WCET analyses are needed to estimate the timing gains due to SPM allocation. After successfully solving the ILP, a final WCET analysis is performed on the optimized program to verify the ILP's prediction. Solving the ILP is an NP-hard problem, and the ILP's solving time rises with the number of variables and constraints. However, the ILP solver can usually quickly determine one first non-integer solution (or prove that no such solution exists). If no non-integer solution exists, then the ILP solver can terminate with an error. Otherwise, most time is spent by the solver to find an *integer* solution for the given problem. For the evolutionary algorithm, a decline in solving times can be noted for eight tasks on eight cores. This stems from the fact that for some benchmarks, the hyper-period was so large that the approach by Xu could not be applied without heavily altering the generated periods. In these cases, the task set was considered as not repairable for both ILP and genetic algorithms. Subsequently, the compilation does not proceed end exits with an error. For the genetic algorithm, this heavily influences the average runtime. For the ILP-based algorithm, the average runtime for six tasks and eight tasks is significantly below the timeout. Thus, these outliers do not have such a large impact on the average solving time.

9 CONCLUSIONS

This article presented how a WCET-aware compiler framework may be extended to optimize both multi-task and multi-core systems at code level. We proposed an ILP-based framework and a genetic approach that allow for schedulability-oriented code optimizations.

The approaches were evaluated using a static instruction SPM allocation for an ARM7TDMI multi-core architecture. We showed that the ILP approach is computationally feasible and is able to repair a high amount of task sets up to an unoptimized system load of 220%. The size of our ILP model grows only linearly with the number of tasks. Despite the exponential worst-case complexity of ILP, solving times are well below two hours, even for eight-core systems with 64 tasks in total. The genetic algorithm outperforms the ILP approach for very small task sets but fails for any larger task sets due to the high computational demand of the WCET analyses required for fitness evaluation.

The ILP model is currently extended towards moving selected data structures into private data SPMs. We will then combine this bus-aware data SPM allocation with the instruction SPM optimization of this article. Furthermore, we will integrate cache-aware optimizations like, e.g., Reference [30] into the multi-task ILP model to keep up with the spread of caches in the area of hard real-time systems. Additionally, we aim at extending the framework to be able to automatically allocate tasks to the respective computing cores from within the ILP framework. The evaluation showed that the time-consuming WCET analyses prove to be a show-stopper of using the genetic algorithm to optimize larger systems. Additionally, due to the fact of being NP-hard, solving times of the ILP-based approach also grow exponentially with the size of the task sets to be optmized. Future research should thus tackle both issues. A promising approach on reducing the time needed by the WCET analysis is to substitute the WCET analysis by a much faster but also more pessimistic WCET estimation. Especially for memory allocations, we consider it to be relatively easy to give a a fair estimate on the *change* of the WCET of an individual for a given reallocation of some basic blocks.

REFERENCES

[1] AbsInt Angewandte Informatik, GmbH. 2019. aiT Worst-Case Execution Time Analyzers. https://www.absint.com/ait/.

[2] Sanjoy K. Baruah. 2003. Dynamic- and static-priority scheduling of recurring real-time tasks. *Springer Real-time Syst.* 24, 1 (2003).

[3] Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the performance of schedulability tests. *Real-time Syst.* 30, 1 (5 2005), 129–154.

[4] Enrico Bini, Marco Di Natale, and Giorgio Buttazzo. 2007. Sensitivity analysis for fixed-priority real-time systems. *Real-time Syst.* 39, 1–3 (2007), 5–30.

[5] Johannes Bisschop. 2017. *AIMMS Optimization Modeling*. AIMMS B.V., Haarlem, The Netherlands.

[6] Corinna G. Lee and Paul Chow and Mark G. Stoodley. 2018. UTDSP Benchmark Suite. Retrieved from www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.

[7] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. 2010. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of the Embedded Real Time Software and Systems Conference*. 36–42.

[8] M. de. Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. 2008. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 161–166.

[9] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A benchmark collection to support worst-case execution time research. In *Proceedings of the Worst-Case Execution Time Analysis Conference*.

[10] Heiko Falk and Jan C. Kleinsorge. 2009. Optimal static WCET-aware scratchpad allocation of program code. In *Proceedings of the Design Automation Conference*.

[11] Heiko Falk and Paul Lokuciejewski. 2010. A compiler framework for the reduction of worst-case execution times. *Springer Real-time Syst.* 46, 2 (2010).

[12] David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning* (repr. with corr. ed.). Addison-Wesley Publishing Company, Inc., Reading, Mass.

[13] Klaus Gresser. 1993. An event model for deadline verification of hard real-time systems. In *Proceedings of the Euromicro Conference on Real-time Systems*.

[14] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET benchmarks—Past, present and future. In *Proceedings of the Worst-case Execution Time Analysis Conference*.

[15] M. Guthaus, T. Austin, D. Ernst, R. Brown, T. Mudge, and J. Ringenberg. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC)*, Vol. 00. 3–14.

[16] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. 1998. Bounding loop iterations for timing analysis. In *Proceedings of the 4th IEEE Real-time Technology and Applications Symposium (Cat. No.98TB100245)*. 12–21.

[17] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. 2005. System level performance analysis—The SymTA/S approach. *IEEE Proc. Comput. Dig. Techniq.* 152, 2 (Mar. 2005), 148–166.

[18] Mathai Joseph and Paritosh Pandya. 1986. Finding response times in a real-time system. *Comput. J.* 29, 5 (1 1986), 390–395. DOI: DOI : https://doi.org/10.1093/comjnl/29.5.390.

[19] Timon Kelter. 2015. *WCET Analysis and Optimization for Multi-core Real-time Systems*. Ph.D. Dissertation. TU Dortmund University, Dortmund, Germany.

[20] T. Kelter, H. Borghorst, and P. Marwedel. 2014. WCET-aware scheduling optimizations for multi-core real-time systems. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS'14)*.

[21] Timon Kelter and Peter Marwedel. 2017. Parallelism analysis: Precise WCET values for complex multi-core systems. *Sci. Comput. Progr.* 133 (2017).

[22] Chunho Lee, M. Potkonjak, and W. H. Mangione-Smith. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th International Symposium on Microarchitecture*. 330–335.

[23] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. 2007. Chronos: A timing analyzer for embedded software. *Sci. Comput. Progr.* 69, 1 (2007), 56–67.

[24] C. L. Liu and James W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* 20, 1 (1973).

[25] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. 2009. A fast and precise static loop analysis based on abstract interpretation, program slicing, and polytope models. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'09)*. 136–146.

[26] Paul Lokuciejewski, Timon Kelter, and Peter Marwedel. 2010. Superblock-based source code optimizations for WCET reduction. In *Proceedings of the 10th IEEE International Conference on Computer and Information Technology*. 1918–1925.

[27] Louis-Noel Pouchet. 2018. PolyBench/C—The Polyhedral Benchmark Suite. Retrieved from http://www.cs.ucla.edu/pouchet/software/polybench/.

[28] Arno Luppold and Heiko Falk. 2015. Code optimization of periodic preemptive hard real-time multitasking systems. In *Proceedings of the IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC'15)*.

[29] Arno Luppold and Heiko Falk. 2017. Schedulability-aware SPM allocation for preemptive hard real-time systems with arbitrary activation patterns. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE'17)*.

[30] Arno Luppold, Christina Kittsteiner, and Heiko Falk. 2016. Cache-aware instruction SPM allocation for hard real-time systems. In *Proceedings of the 19th International Workshop on Software & Compilers for Embedded Systems (SCOPES'16)*. 77–85.

[31] Arno Luppold, Dominic Oehlert, and Heiko Falk. 2018. *Evaluating the Performance of Solvers for Integer-Linear Programming*. TR. Institute of Embedded Systems, Hamburg University of Technology, Hamburg/Germany.

[32] G. Memik, W. H. Mangione-Smith, and W. Hu. 2001. NetBench: A benchmarking suite for network processors. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD'01). IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*. 39–42.

[33] Milica Mitić and Mile Stojčev. 2006. A survey of three system-on-chip buses: AMBA, CoreConnect, and wishbone. In *Proceedings of the International Scientific Conference on Information, Communication and Energy Systems and Technologies*. 282–285.

[34] Moritz Neukirchner, Sophie Quinton, Tobias Michaels, Philip Axer, and Rolf Ernst. 2013. Sensitivity analysis for arbitrary activation patterns in real-time systems. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE'13)*. 135–140.

[35] Dominic Oehlert and Heiko Falk. 2018. WCET analysis of automotive buses using WCC. In *Proceedings of the DATE Workshop on New Platforms for Future Cars*.

[36] Dominic Oehlert, Arno Luppold, and Heiko Falk. 2016. Practical challenges of ILP-based SPM allocation optimizations. In *Proceedings of the Software and Compilers for Embedded Systems (SCOPES'16)*.

[37] Dominic Oehlert, Arno Luppold, and Heiko Falk. 2017. Bus-aware static instruction SPM allocation for multicore hard real-time systems. In *Proceedings of the Euromicro Conference on Real-time Systems (ECRTS'17)*.

[38] Guillaume Phavorin, Pascal Richard, Joël Goossens, Thomas Chapeaux, and Claire Maiza. 2015. Scheduling with preemption delays: Anomalies and issues. In *Proceedings of the International Conference on Real-time and Network Systems (RTNS'15)*. 109–118.
[39] Kai Richter. 2005. *Compositional Scheduling Analysis Using Standard Event Models. The SymTA/S Approach*. Ph.D. Dissertation. Technical University Carolo-Wilhelmina of Braunschweig.
[40] StreamIt Community. 2018. The StreamIt Benchmark Suite. Retrieved from http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml.
[41] V. Suhendra, T. Mitra, A. Roychoudhury, and Ting Chen. 2005. WCET centric data allocation to scratchpad memory. In *Proceedings of the IEEE Real-time Systems Symposium (RTSS'05)*.
[42] Synopsys Inc. 2018. CoMET System Engineering IDE. Retrieved from http://www.synopsys.com.
[43] Alan M. Turing. 1937. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.* s2-42, 1 (1937), 230–265. DOI: https://doi.org/10.2307/2268810
[44] Ernesto Wandeler. 2006. *Modular Performance Analysis and Interface-based Design for Embedded Real-Time Systems*. Ph.D. Dissertation. Swiss Federal Institute of Technology Zurich, Zurich/Switzerland.
[45] Jia Xu. 2010. A method for adjusting the periods of periodic processes to reduce the least common multiple of the period lengths in real-time embedded systems. In *Proceedings of the IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA'10)*. 288–294.
[46] Fengxiang Zhang, Alan Burns, and Sanjoy Baruah. 2011. Sensitivity analysis of arbitrary deadline real-time systems with EDF scheduling. *Real-time Syst.* 47, 3 (2011), 224–252.
[47] Marco Ziccardi, Alessandro Cornaglia, Enrico Mezzetti, and Tullio Vardanega. 2015. Software-enforced interconnect arbitration for COTS multicores. In *Proceedings of the Worst-case Execution Time Analysis Conference (WCET'15)*.
[48] Vojin Zivojnović, Juan M. Velarde, Christian Schläger, and Heinrich Meyr. 1994. DSPSTONE: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology (IC-SPAT'94)*.

Received January 2019; revised October 2019; accepted December 2019