ETH Library

A Synergistic Approach to Predictable Compilation and Scheduling on Commodity Multi-Cores

Conference Paper

Author(s):
Forsberg, Björn; Mattheeuws, Maxim; Kurth, Andreas; Marongiu, Andrea; Benini, Luca

Publication date:
2020-06

Permanent link:
https://doi.org/10.3929/ethz-b-000420966

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
https://doi.org/10.1145/3372799.3394369

Funding acknowledgement:
871669 - A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimisation (EC)

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

A Synergistic Approach to Predictable Compilation and Scheduling on Commodity Multi-Cores

BJÖRN FORSBERG, D-ITET, ETH Zürich, Switzerland
MAXIM MATTHEEUWS, D-ITET, ETH Zürich, Switzerland
ANDREAS KURTH, D-ITET, ETH Zürich, Switzerland
ANDREA MARONGIU, DPMI, University of Modena, Italy
LUCA BENINI*, D-ITET, ETH Zürich, Switzerland

Commodity multi-cores are still uncommon in real-time systems, as resource sharing complicates traditional timing analysis. The Predictable Execution Model (PREM) tackles this issue in software, through scheduling and code refactoring. State-of-the-art PREM compilers analyze tasks one at a time, maximizing task-level performance metrics, and are oblivious to system-level scheduling effects (e.g. memory serialization when tasks are co-scheduled). We propose a solution that allows PREM code generation and system scheduling to interact, based on a genetic algorithm aimed at maximizing overall system performance. Experiments on commodity hardware show that the performance increase can be as high as 31% compared to standard PREM code generation, without negatively impacting the predictability guarantees.

CCS Concepts: • **Computer systems organization** → **Real-time systems**.

Additional Key Words and Phrases: Real-time Embedded Systems, Predictable Execution Model, Compilers, Scheduling, Optimization

ACM Reference Format:
Björn Forsberg, Maxim Mattheeuws, Andreas Kurth, Andrea Marongiu, and Luca Benini. 2020. A Synergistic Approach to Predictable Compilation and Scheduling on Commodity Multi-Cores. In *Proceedings of the 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '20), June 16, 2020, London, United Kingdom.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3372799.3394369

1 INTRODUCTION

Over the past decade, multi-core systems have taken over every market segment, but their adoption is still slow in the context of real-time systems because contention on shared resources leads to unpredictable access times. The most severely contended resource is the global memory [12], e.g., the DRAM, from which all cores load instructions and data. To guarantee that timing constraints are never violated, the worst case execution time (WCET) of each task is analyzed. On traditional single-core systems, such analysis is well understood and mature tools exist [16]. However, for WCETs valid under any multi-core execution the maximum interference would have to be assumed for every access [17], leading to very pessimistic bounds. These may even nullify the benefits of multi-core execution in the first place as memory latency increases linearly with the number of cores [3].

The Predictable Execution Model (PREM) [11] has been proposed to solve the contention problem by enforcing mutual exclusive access to the shared memory system. To avoid stalling other tasks while memory is occupied, the tasks

*Also with University of Bologna, Italy.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

are transformed to utilize their allotted memory time to copy data from global memory to core private *local memory*. As local memory is not subject to interference, computation can be performed without contention. To achieve this, PREM tasks are divided into *intervals* of *prefetch* (P), *compute* (C), and *writeback* (WB) phases. The C phase performs the local computation, the P phase brings data to the local memory, and WB moves data back to global memory. Only the P and WB phases access memory, and are referred to as *memory* phases. By scheduling the system such that only a single task is executing a memory phase at a time, interference can be effectively prevented, and the WCET can be calculated in isolation. When the system is composed of several tasks, the timing properties of individual tasks remain the same, but time intervals spent waiting for memory access need to be taken into account. Thus, PREM transforms the interference problem into a statically solvable scheduling problem [1, 9, 18]. Besides scheduling, another important aspect is the proposal of PREM compilers [9, 14]. These automate the transformation of programs into sequences of P, C, and WB phases, a tedious and error-prone process.

However, PREM compilers and schedulers are individually unable to construct a well-optimized PREM system. The compiler has a local view of each task, which it can optimize for predictability and performance, but it cannot optimize for the interactions with other tasks deployed on the system. This is done by the scheduler, which has full visibility of all tasks in the system. However, even a state-of-the-art optimal scheduler may produce sub-optimal schedules, as the scheduler cannot improve the schedule beyond the degrees of freedom given by the intervals created by the compiler.

Furthermore, the objectives of the scheduler and compiler optimizations are often in direct conflict: Each individual task will perform better if larger intervals are selected, as this reduces the cost of scheduling at the PREM phase boundaries. The scheduler, in contrast, has the most amount of freedom if PREM intervals are selected as small as possible, as blocking memory phases can then be interleaved at finer granularity. Thus, for each PREM system there exists an optimum where for per-task and per-system objectives are combined, but it can not be found by the compiler or scheduler in isolation.

Soliman et al. [15] address this concern by integrating the scheduler into the compiler, but this solution is still subject to the local view of the compiler: All tasks need to be compiled together in a single compilation unit to enable scheduling, breaking common development flows such as partial compilation into object files. To fully address this issue, a new methodology that preserves the strengths of both the compiler's per-task and scheduler's per-system optimization, while finding the sweet spot is required. This work addresses this question, making the following contributions:

- A novel methodology for optimizing PREM systems.
- An implementation of the new tools needed to realize the methodology in practice.
- Results showing that this methodology can reduce the response time of PREM systems by as much as 31%.

The paper is structured as follows: Section 2 provides the background on PREM compilation and Section 3 defines the problem this paper addresses. Following this, Section 4 provides the proposed solution, which is evaluated in Section 5. Section 6 presents related work and Section 7 concludes.

2 BACKGROUND

This section describes the underlying architectural problem from a real-time perspective, and how PREM solves it. Following this, we provide an overview of the features of PREM compilers and PREM schedulers relevant to this work.

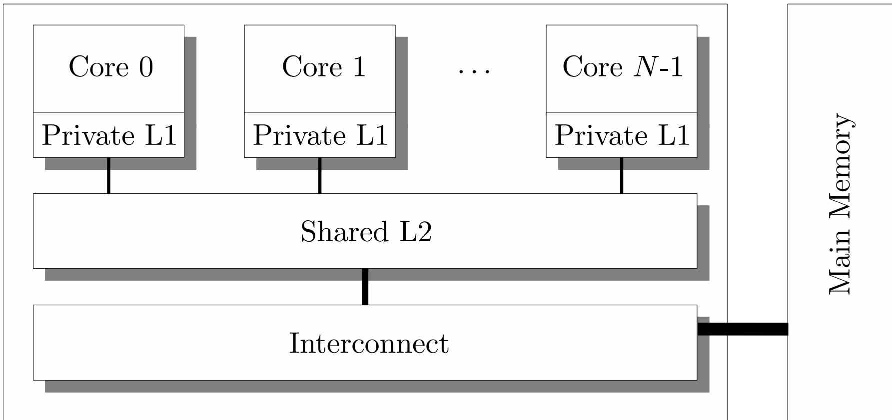


Fig. 1. An overview of the architectural template.

2.1 Architectural Template and Interference

As shown in Figure 1, our work considers a multi-core system consisting of N cores $\{c_0, c_1, \cdots, c_{N-1}\}$, each with their own private L1 cache, which we refer to as the local memory L. Additionally the system may share zero or more layers of shared caches, illustrated in the figure with a shared L2 cache. The last level cache (LLC) is connected via an on-chip interconnect to an off-chip memory, e.g., DRAM.

With the exception of the private L1 memory, the entire memory system is shared in multi-core systems. This resource sharing means that the latency to access memory from one core depends on the memory activity of other cores. While this is not a problem in general purpose systems, it makes the deployment of real-time systems problematic. The worst case execution time (WCET) is required for schedulability analysis and provably safe operation, but variance in the memory access latency makes the analytical computation of the WCET difficult, if not infeasible.

2.2 The Predictable Execution Model

PREM solves the contention problem by only permitting a single core c at a time to access the global DRAM. This *mutually exclusive* access to the shared memory ensures that memory latency can be modeled in *single-core equivalent* terms, i.e., as if the memory system was not shared with other cores [11], and the WCET can be calculated in isolation. Thus, only the blocking time while waiting for the *memory mutex* needs to be taken into account, transforming the interference problem in multi-core systems into a scheduling problem which we will discuss in Section 2.4.

The system consists of a number of tasks $\tau \in T$, where T is the set of all tasks to be executed on the system. Each task τ is mapped to a core c, and for the purposes of this paper we assume that there is no migration at runtime. To achieve mutually exclusive memory accesses the Predictable Execution Model divides each task $\tau \in T$ into a sequence of *intervals* $I_\tau = \{i_0, i_1, \cdots, i_n\}$. Each interval i internally consists of independently schedulable prefetch (P), compute (C) and writeback (WB) phases, where the P and WB phases are referred to as the memory (M) phases. The memory phases are responsible for moving the data from the shared memory to a core-private memory L which is not subject to interference, upon which the C phase computes. Importantly, this means that only the memory phases P and WB need to be scheduled with mutually exclusive memory access. To ensure that all data can be stored locally, the size of the data accessed within an interval $size(i)$ must be dimensioned such that it is smaller than the size of the local memory

```
void main() {
  int A = 0;
  for (int i = 0; i < 100; i++) {
    A = A + B[i]
  }
  return A;
}
```

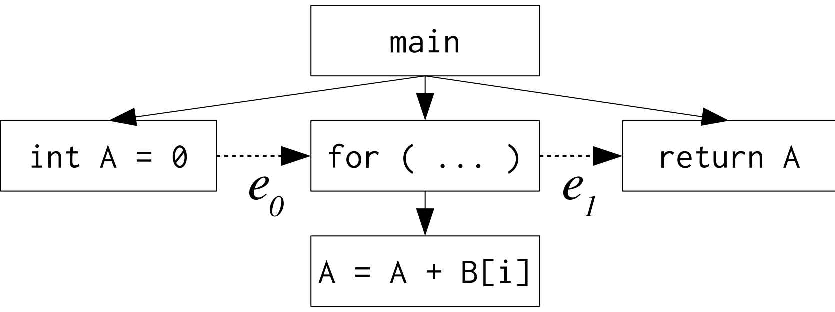


Fig. 2. An example program and the resulting region tree, with control flow edges e overlaid.

$size(L)$, as shown in Equation 1.

$$\forall \tau \in T : \forall i \in I_\tau : size(i) < size(L) \tag{1}$$

There exist multiple valid partitionings of a task τ into intervals I_τ, the selection of which is the task of the compiler.

2.3 PREM Compilers

Due to the complexity of making programs PREM-compliant, compiler support has been proposed [9, 14] to automatically identify code segments that fulfil the requirements of Equation 1, and transform these segments into PREM *intervals* of *prefetch*, *compute*, and *writeback phases*. The following section outlines three of the major components of PREM compilers [9, 14] that impact the final PREM system.

All recently proposed PREM compilers [4, 9, 14] use Single Entry Single Exit (SESE) regions as the atomic unit from which PREM intervals are created. SESE regions are code regions represented by every part of the Control Flow Graph (CFG) that only has a single incoming edge and a single outgoing edge. This maps well to PREM intervals, as data can be loaded on the incoming edge and stored on the outgoing edge. Regions are hierarchical, and are represented as a tree Υ_τ, as shown in Figure 2: Each SESE region may contain SESE regions within them, but not across the boundaries of the parent region. The solid arrows represent the parent-child relationship of the tree property. For non-loop regions the union of all children make up the same code as the parent region. For loop-regions the parent region can be unrolled into a sequence of their child regions. In addition to the tree property, the control flow graph of the program is overlayed on the table with dotted arrows, representing the SESE edges e between the regions in the program. For a detailed

description of the region tree we direct the interested reader to [14], that also describe how to use this interprocedurally. We refer to the set of all regions in task τ as Υ_τ.

For the compiler to create intervals that respect the requirement of Equation 1, the memory footprint (i.e., size in bytes of data accessed) of all regions must be calculated. This can be calculated as outlined in [9, 14]. For the remainder of the discussion we assume this methodology is used.

With each region $r \in R_\tau$ annotated with its memory footprint, the compiler can assign each region into a PREM interval I, after which the code is transformed into PREM memory and compute phases based on these intervals.

As outlined previously, there are two fundamental types of regions: sequential regions and loops [9]. The selection of sequential regions r_{seq} into an interval I is a binary decision: Either r_{seq} is in an interval $i \in I$ or it is not. If a sequential region is too large to fit into a PREM interval, its children in the region tree can be selected, or if it is a leaf node, it can be split. Loops smaller than the local memory can be treated in the same way. The only feasible approach for loops too large to fit into a PREM interval is to *tile* them. Tiling is typically used to increase cache locality and involves dividing a set of large loops into a larger set of smaller loops, thus reducing the memory footprint of each inner loop. Loops are represented in the region tree as loop regions r_{loop}, and their tiling can be expressed as a tuple (r_{loop}, g), where g is the number of iterations in the new nested loop, referred to as the tiling granularity. We return to this in Section 4.2.

2.4 PREM Schedulers

The objective of PREM scheduling is to ensure that *memory interference* is effectively avoided, while still ensuring that all tasks τ meet their deadlines D_τ. Memory interference is avoided by finding a system schedule that maps each interval i to a core c, and globally scheduling the system such that only a single core c is executing the memory phase of an interval i at a time. Scheduling techniques to achieve this are readily available in the literature [1, 9, 18], and as all share the fundamental requirement that only one task is executing its memory phase at once, the total response time R_τ of a task τ can be generically modeled as shown in Equation 2.

$$R_\tau = B^{core} + B^{memory} + S(|I_\tau|) + e_\tau \tag{2}$$

Here, B^{core} is the *blocking time* due to core-local scheduling, e.g., the increase in the response time due to τ being preempted by another task executing on the same core. The B^{memory} term is the *blocking time* due to a τ having to wait for a task on another core using the memory, due to the *mutually exclusive* policy at the heart of PREM. The S term is the static cost of performing the context switch for performing the online scheduling decision. This cost may vary from small (e.g., cost of a function call to determine the next interval in a pre-computed static schedule) to very large (e.g., a *syscall* and online decision from a dynamic scheduler). This cost grows linearly with the number of intervals $|I_\tau|$ in τ that require handling during execution [5]. The specific scheduling policy (e.g., fixed priority, earliest deadline first, etc.) determines when a task is blocked. Lastly, the e_τ term is the accumulated worst case execution time of all intervals in task τ, as shown in Equation 3.

$$e_\tau = \sum_{i \in I_\tau} len(i) \tag{3}$$

Here, $len(i)$ is the worst case execution time (WCET) of interval $i \in I_\tau$. For the remainder of this discussion, we will assume that $len(i)$ is provided by an external tool which we will refer to as the *WCET analyzer*, of which many have been proposed in the literature [16]. As PREM scheduling implies *single-core equivalence* for the WCET analysis, classical single core analysis techniques can be used. As is customary, we say that a taskset T is schedulable if every task in the

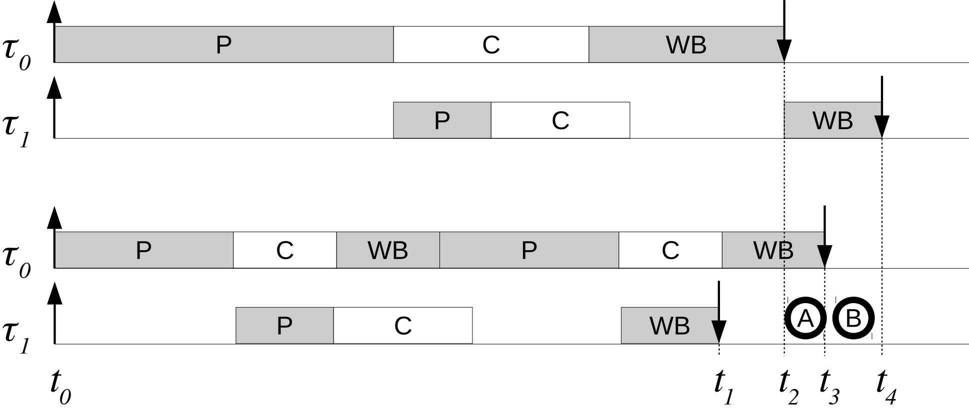


Fig. 3. Illustrative example of how PREM interval sizing of individual tasks affects the overall system performance.

taskset responds before its deadline, as shown in Equation 4.

$$\forall \tau \in T : R_\tau < D_\tau \tag{4}$$

For the remainder of the discussion, we will only consider a single task τ executing per core c, and as such the term B^{core} will always be zero, assuming $|T| \leq N$. However, the fundamental insights of this paper generalize to the case where multiple tasks are deployed on each core, although the relative impact of B^{core} on the remaining terms may lead to a different optimal configuration. Following this, we revise Equation 2 as shown in Equation 5.

$$R_\tau = B^{memory} + S(|I_\tau|) + e_\tau \tag{5}$$

We use the notation $R_{\tau_0,\tau_1,\dots}$ to refer to the total response time of the system, defined as the maximum response time of any of the tasks in the system $max(R_{\tau_0}, R_{\tau_1}, \dots)$.

3 PROBLEM DESCRIPTION

As shown in Equation 5, there are three terms that should be minimized to reduce the response time R_τ, thus both improving the performance of the system, and increasing the likelihood of making the taskset T schedulable. The accumulated interval WCET e_τ is ideally constant[1], but the remaining terms can be tuned: To minimize B^{memory}, the interval lengths $len(i), i \in I_{\bar{\tau}}$ (where $\bar{\tau}$ means tasks other than τ) should be *minimized* to reduce blocking time. On the other hand, to minimize $S(|I_\tau|)$ the interval lengths $len(i), i \in I_\tau$ should be *maximized* to reduce the number of scheduling points and their overheads. We make two key observations:

Observation 1: Both options to alter the intervals $i \in I_\tau$ are only available during compilation. The first is to alter $size(i)$, as a smaller interval size (in bytes) will lead to less computation in the interval i, and a shorter execution time $len(i)$. The other option is to completely alter the scheduling conditions by selecting a different set of intervals I' altogether.

Observation 2: The impact of both options on R_τ is only known after scheduling, and cannot be used by the compiler to select a different I' or to alter $size(i), i \in I$. Furthermore, the compiler can only infer information about the task τ

[1]In practice, intervals that contain loop tiles may incur non-negligible tiling overhead for small loop sizes, but at that point $S(|I_\tau|)$ will dominate.

under compilation, limiting optimizations to criteria available at the granularity at which compilers analyze programs (i.e., translation unit), effectively excluding optimizations to $\bar{\tau}$.

In light of these limitations, previously proposed PREM compilers [9, 14] have fallen back to select PREM intervals that are as large as possible, while still fitting into the local memory. This reduces the total amount of online scheduling decisions at runtime, and therefore $S(|I_\tau|)$, at the cost of increasing B^{memory} for other tasks. Thus, the current state of the art compilers may, by reducing one of the terms, inadvertently increasing the other, overall worsening R_τ.

Consider the example in Figure 3. In the top scenario, tasks τ_0 and τ_1 have been compiled with the larger-is-better interval sizing heuristic. While this is best for the performance of each individual task, the co-scheduled system suffers from serialization effects, as the memory phases (P, WB) of τ_1 cannot be co-scheduled with those of τ_0. In this example, both tasks are released at time t_0 and the final phase of τ_1 finishes at t_4, which is the response time $R_{\{\tau_0,\tau_1\}}$ of the taskset $\{\tau_0, \tau_1\}$. In the bottom scenario, τ_0 has been divided into intervals of half the size. Due to scheduling overhead, the task now takes longer to execute, finishing at t_3 instead of t_2, marked Ⓐ. However, due to the finer scheduling granularity, τ_1 can now be scheduled earlier, completing already at t_1. Thus, while we increased the execution time of τ_0, the response time of the taskset is reduced from t_4 to t_3, marked Ⓑ.

Clearly, there exists a trade-off between the best performance for the individual tasks and the best performance for the overall system. This could potentially be addressed by in-compiler techniques (e.g., link-time optimization), but no approach exists that allows the entire set of schedulable tasks to be put under the control of the compiler. Instead, we propose a novel methodology for PREM system deployment.

4 SYNERGISTIC OPTIMIZATION METHODOLOGY

For the first time, our methodology enables PREM tools, i.e., compiler, WCET analyzer, and scheduler to exchange information, as shown in Figure 4, to globally optimize the system. It can be fully automatized, driven by the novel *Optimizer* component. By using this methodology, the source code and real-time constraints are automatically transformed, analyzed, and scheduled to produce an optimized PREM system, outputting executable binaries and a static system schedule.

The first steps of the methodology are directly derived from the traditional PREM system deployment approach: The first step of the proposed method is the compilation of the source code for each PREM task $\tau \in T$ with the PREM compiler to produce the PREM intervals I. As a starting point, the compiler relies on a simple heuristic that maximizes use of available local memory (to minimize synchronization overheads, $S(|I_\tau|)$). The compiled program is analyzed for WCET, i.e., upper bounds of the execution time of each PREM intervals $len(i)$ are derived. This can be done through measurements or using one of the static analysis methods available in the literature [16]. This provides the input to the PREM scheduler [1, 9, 18] together with the real-time constraints, e.g., the deadline D_τ. Additionally, as proposed by Matejka et al. [9], the scheduler takes a directed acyclic graph (DAG) of the ordering of intervals as input, which is produced by the PREM compiler. The DAG specifies the dependencies between the PREM intervals to preserve program order in the constructed schedule. At the top level, the SESE edges e (see Section 2.3) that connect the PREM intervals are transferred to the DAG, and within each intervals there are edges that specify the dependency of the WB phase on the C phase, and of the C phase on the P phase. The scheduler also gives additional information, e.g., a binary fail/success if the schedule respects the schedulability constraint as given in Equation 4, and the response time R of the system. This is where the traditional approach would end.

However, as outlined in Section 3, the identified solution may not be optimal with respect to R, as the schedule is only optimal for the exact set of intervals I produced by the compiler. If all real-time constraints are met (Equation

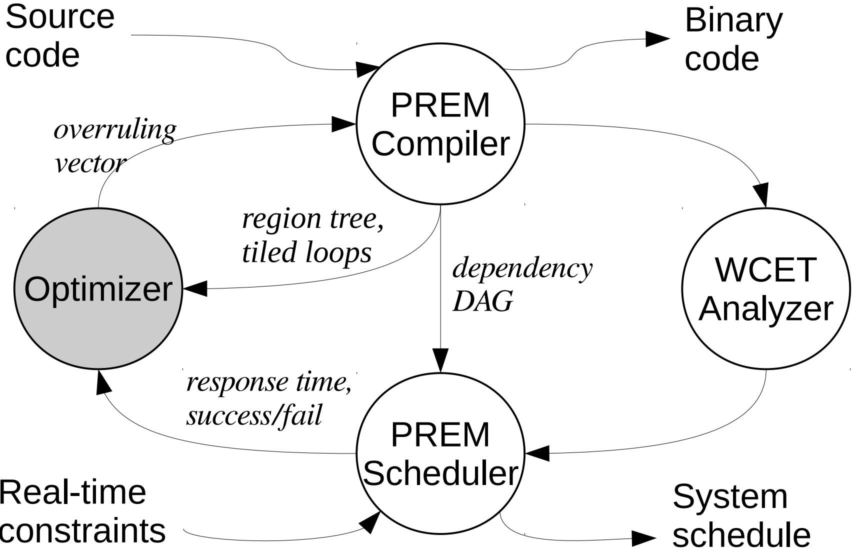


Fig. 4. Data exchange in the proposed methodology.

4), this might be acceptable, but if the scheduler was unable to schedule the selected intervals I without violating the constraints, this does not necessarily mean that the system is unschedulable under PREM.

4.1 Connecting the Region and Time Domains

This is where our novel methodology offers a solution, which enables the exchange of high-level information between the scheduling and compilation steps to enable a synergistic optimization of the system.

Importantly, and as outlined in Sections 2.3 and 2.4, the compiler operates in the *region domain* of the source code of the program, while the scheduler operates in the *time domain*. However, it is inadvisable to intertwine the fundamentally different *region* and *time domains* of the compiler and scheduler, as this increases tool complexity and interdependence, increasing the development and maintenance cost of either tool. In particular, the compiler should not be extended to be aware of the behavior of tasks outside its current compilation unit, and the scheduler should not be extended to understand the low-level code details of SESE regions in the *region domain*. The only shared concept that exist between the compiler and the scheduler are the PREM intervals I, which we instead use to pass information between the two.

To enable this, we introduce a new component, the *Optimizer* as shown in Figure 4 to manage the additional information. This optimizer is responsible for translating the output in the *time domain* from the scheduler, into a refined interval selection based on the regions Υ_τ of each task τ – thus ensuring that the compiler only handles the τ in the current translation unit.

As shown in Figure 4, the PREM scheduler retains the same input and output parameters as defined in Sections 2.4 and 4, but the output is in the proposed methodology redirected to the *optimizer*, which triggers a re-compilation and re-scheduling with a different set of intervals I' based on an *overruling vector* passed to the compiler. The *overruling vector* is given in terms of regions $r \in \Upsilon_\tau$ that the compiler natively understands, allowing it to alter the I_τ selected to improve the response time R achieved by the scheduler.

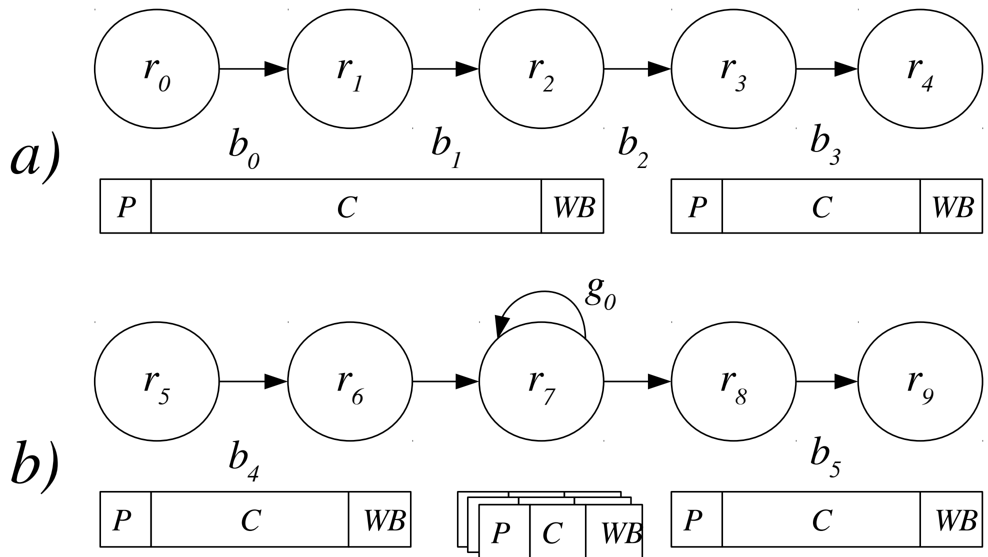


Fig. 5. Overruling parameters linked to regions.

We will describe how the *optimizer* achieves this translation in Section 4.3, but first we define *overruling vectors*.

4.2 Overruling Vectors

The new *overruling vector* overrules the internal heuristics for interval selection in the compiler. We define the overruling vector Z as a collection of boolean values b, where each b represents a decision whether two regions $r_0, r_1 \in \Upsilon_\tau$ are to be combined into the same interval. Each b is mapped to the unique SESE edge e_{r_0,r_1} (see Section 2.3) connecting the SESE regions r_0 and r_1 in the CFG. If the boolean value b is *true* both regions r_0 and r_1 connected by this edge will be selected into the same interval i. Consider the example in Figure 5a: Five sequential regions r_0, r_1, r_2, r_3, r_4 are shown in the CFG. The figure depicts the interval selection that $\{b_0, b_1, b_2, b_3\} = \{1, 1, 0, 1\}$ would result in, giving two intervals $i_0 = \{r_0, r_1, r_2\}, i_1 = \{r_3, r_4\}$. This corresponds to the selection of regions r_{seq} in Section 2.3, which uses compiler heuristics to perform the same process. However, the overruling vector allows the proposed methodology to alter the interval selection of the built-in compiler heuristics to improve the overall system response time of all tasks, as we outlined in Section 3.

Regions with loops r_{loop} that are larger than local memory $size(r_{loop}) > size(L)$ cannot be joined with other regions into an interval, as the loop region in itself is already too large, thus violating the constraint in Equation 1. This can intuitively be addressed by unrolling the loop in region r_{loop} into several smaller regions $r_{loopchunk_0}, r_{loopchunk_1}, \cdots$ (see Section 2.3), and selecting each of them using the boolean values b for the resulting edges $e_{r_{loopchunk_0}, r_{loopchunk_1}}$ that connect them. However, previously proposed PREM compilers [9] have built-in support for *loop tiling*, achieving the same effect but without unrolling the code and increasing the code size. In this case, the compiler will see a single region r_{loop}, and to inform the compiler that this region is to be split into tiles, we introduce the additional overruling vector L of tuples (r, g). Every tuple selects the *tiling granularity* g for the loop region r, determining how many iterations of

the loop that are executed within each tile, as shown in Figure 5b. This constitutes a compact way of representing the unrolling operation that would otherwise be required.

Note that regions r_{loop} that are tiled can not be selected into an interval i with their predecessor and successor regions r_{pred} and r_{succ}. This, as the interval would without tiling violate the size constraint in Equation 1, and tiling invalidates the edges $e_{r_{pred},r_{loop}}$ and $e_{r_{loop},r_{succ}}$ (replacing them with multiple new edges). Therefore we do not associate a boolean value $b \in Z$ with edges connecting to a loop that must be tiled, allowing only the use of $(r, g) \in L$, as shown in Figure 5b. Here r_5 and r_6, as well as r_8 and r_9 retain the $b \in Z$ vector, while r_7 is only associated with $(r, g) \in L$. Note that the loop back-edge for region r_7 is placed outside the region for illustration purposes only. Note also that loop regions r_{loop} which are smaller than the local memory $size(r_{loop}) \le size(L)$ do not need to be tiled, but can be handled with the Z vector.

To guarantee that the resulting intervals from the overruling vectors L and Z still correspond to correct PREM intervals, we extend the compiler to validate that the overruled interval selection do not violate the constraint in Equation 1.

4.3 Optimizer

The new *optimizer* triggers, as part of the novel methodology, a re-scheduling with a different set of intervals I' based on the *overruling* parameters. This can be executed for any number of iterations, either until an I that results in a feasible schedule is found, or further optimized according to some additional metric. Because the compiler selected the optimal intervals for each task τ during the initial compilation (optimizing $S(|I_\tau|)$), this implies that we are trading off per-task performance to produce smaller intervals that enable finer-grained scheduling to optimize B^{memory}.

To construct I', the *optimizer* constructs the necessary *overruling vectors* based on the result from the previous scheduling. To achieve this, the compiler exports the regions $r \in \Upsilon_\tau$ as an XML file. This allows the optimizer to deconstruct a previous interval $i \in I_\tau$ into its constituent regions $r \in \Upsilon_\tau$ (required for the *overruling vectors*), and reassemble them into any number of new intervals $i' \in I'_\tau$.

Depending on the amount of metadata produced by the scheduler, the selection of which compiler decisions to *overrule* can be differently precise: If the scheduler outputs information about which intervals are blocking which, these can be selectively overruled, but in the generic case when no such information is provided, the *optimizer* must use a suitable algorithm to determine how to produce *overruling* vectors. We will discuss this further in Section 5.3, but first present a generic optimizer that makes as few assumptions as possible on the output of the scheduler.

4.3.1 Genetic PREM Optimizer: A Use Case. While the *optimizer* can be tailored to the unique characteristics of a specific PREM scheduler, we propose an *optimizer* that uses only the *response time R* of the system (a quantity that any PREM scheduler will output) to implement the proposed methodology. As small changes in the interval sizing could potentially significantly alter the optimal interleaving of memory phases, we are searching for a global optimum on a non-continuous optimization function. For this reason, we base our optimizer on a genetic algorithm (GA) [6], as they are known to perform well on such functions [2]. GAs operate on *populations* of N *individuals* and execute in epochs. At the end of each epoch the *fitness F* of the population is evaluated, in our algorithm given by $F = \frac{1}{R}$, where R is the response time. Each individual in the population has a different *genome* γ that represents the characteristic of the individual. In our algorithm, the genome is represented by the *overruling vectors* Z and L for every task in the taskset as follows: Each $b \in Z$ is a single bit representing the decision whether two regions are to be combined into the same interval. Each $g \in L$ represents the tiling granularity and uses as many bits as required to express the maximum

Table 1. Benchmarks of different memory complexity.

Benchmark	Description	Complexity		Shorthand
		Compute	Memory	
axpy	Vector addition	n	n	AY
gemv	Matrix-vector mult.	n^2	n^2	GV
gemm	Matrix-matrix mult.	n^3	n^2	GM
jacobi-1d	1D Jacobi stencil	n	n	JA
conv-2d	2D Convolution	n^2	n^2	C2
conv-3d	3D Convolution	n^3	n^3	C3

tiling granularity that results in an interval smaller than the local memory. This value is provided by the compiler. As such, we can represent the parameters L and Z, which describe each possible interval selection of the task (e.g., all *individuals*), as a single binary string, which maps well to GAs. Reusing the example from Figure 5, example (a) would use four bits to represent b_0, b_1, b_2, b_3, and example (b) would use two bits to represent b_4, b_5, as well as the bits required to represent g_0. If the maximum legal value of g_0 is 432, an additional 9 bits would be used to represent $base_2(g_0) = 110110000$. Any g not on the form $2^n - 1$ can express a tiling granularity that would result in intervals larger than $size(L)$, violating Equation 1, and we assign these a fitness score of 0.

At the end of each epoch *individuals* with the worst fitness F are eliminated, determined by the generation gap parameter G. We use $G = 0.5$, meaning 50% of the individuals are eliminated. Following this, new *individuals* are generated through two processes: *Crossover*, in which the genome of two surviving individuals are combined into a new individual, and *mutation*, where a new individual is generated by randomly changing the genome of an individual [6]. The former explores solutions close to known good solutions as well as their linear combinations, and how many individuals are affected is determined by the crossover rate C. The latter introduces random variance into the genome by randomly flipping a bit in the genome γ so that the algorithm is not stuck in a local optimum. How often this occurs is determined by the mutation rate M.

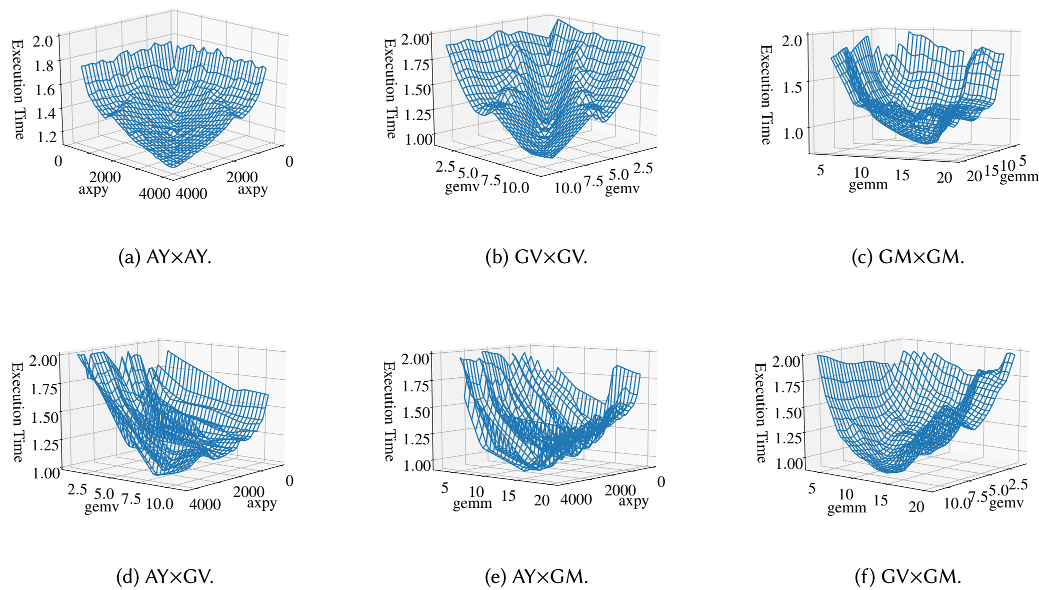
5 EVALUATION

We evaluate the performance of PREM tasksets generated by the proposed methodology, showing it is able to improve the PREM interval selection such that inter-task memory blocking B^{memory} is minimized. We show that reducing the performance of one or more tasks $\tau \in T$ can lead to a better overall response time R of the system. We also show that this requires information that is not available at compile time, motivating the need for the proposed methodology.

To investigate memory blocking, we use benchmarks of different compute-to-communication ratios (CCR), which block the memory for different amounts of time. We initially consider Basic Linear Algebra Subroutines (BLAS) [8], which are classified according to their CCR, as shown in the first three rows of Table 1: The memory-bound BLAS1 kernel *axpy*, the BLAS2 kernel *gemv*, and the compute-bound BLAS3 kernel *gemm*. We refer to these kernels as AY, GV, and GM.

We use the optimizer from Section 4.3.1 and implement *overruling vectors* in previously presented compilers [9, 14]. Evaluation of PREM schedulers and WCET analyzers is out of scope of this work, and we use a first-come-first-served (FCFS) schedule and measurement-based execution times. As the benchmarks are loop-based, they always execute in a steady-state of repeating fixed sized intervals under FCFS.

We implement PREM runtime scheduling in a separate process, which we call the *memory arbitrator* (MA), on one of the unused cores. The MA ensures that only a single task executes a memory phase at a time. At the start of each experiment, the PREM tasks use a UNIX socket to connect to the MA to set up a shared memory region *shm* (not



(a) AY×AY.

(b) GV×GV.

(c) GM×GM.

(d) AY×GV.

(e) AY×GM.

(f) GV×GM.

Fig. 6. Execution times of the *symmetric* and *asymmetric* scenarios.

included in measurements). Shared memory is the fastest inter-process communication in the system[2], but still subject to overheads, as will be shown in Section 5.2. Each PREM phase starts with a handshake with the MA using *shm*, and only when the arbitrator gives permission the task executes the phase.

5.1 Evaluation Platform and Setup

We evaluate on the NVIDIA TX2 [10] SoC, featuring a 4-core ARM A57 cluster, each core with a 32 KB private cache, and a 2 MB shared L2. All cores share the global memory. For our PREM setup, we use the A57 cores, dimensioning the PREM intervals to stage data through the private L1. The TX2 runs Ubuntu Linux 16.04, and to avoid OS scheduling interference, we migrate all processes that are not under investigation to a single core. As the tasks are mostly sleeping, their impact on the memory system is negligible.

We execute a PREM task on each of the two remaining cores. For each task τ executed in the experiments (i.e., an instance of AY, GV, or GM), we select input sizes such that each task has the same execution time R_{base}. This provides an intuitive measure of how well the memory blocking time B^{memory} is minimized: If we co-run two tasks that in isolation require R_{base} time units to finish, the time to run the two tasks τ_0 and τ_1 in parallel R_{corun} (i.e., the response time of the taskset) would be the same $R_{corun} = R_{base}$ time units if the tasks never block each other. We can then quantify the optimality score OS of the interval selection as $OS = \frac{R_{base}}{R_{corun}}$. If the tasks never block each other, we would get $OS = 1.0$, while complete serialization gives $OS = 0.5$. For our experiments we select input sizes such that for each task $R^{\tau}_{base} \approx 0.8s$, and for each taskset $R^{\tau_0,\tau_1}_{base} = max(R^{\tau_0}_{base}, R^{\tau_1}_{base})$.

[2]Found using `ipc-bench` (github.com/goldsborough/ipc-bench).

We use the notation $\tau_0 \times \tau_1$ to designate a *scenario* where τ_0 and τ_1 are executed on one core each. We divide the set of scenarios into two classes: *Symmetric scenarios* where both tasks execute the same program but with different genomes, and *asymmetric scenarios* where each task executes a different program. These former scenarios are AY×AY, GV×GV, and GM×GM, and the latter are AY×GV, AY×GM, and GV×GM. To execute the scenarios, we release both tasks simultaneously, measuring the time until both have completed, giving the total response time R_{corun} of the scenario.

5.2 Solution space exploration

For each scenario we plot the execution times for the exhaustive exploration of the selection space in Figure 6, using the *tiling granularity* g of the BLAS kernel loop to represent the selection space of each task. As the benchmarks are loop-bound, the effect of the interval *overruling* parameter L will be most dominant, and we will discuss the impact of the Z vector in Section 5.2.1. As outlined in Section 3, the compiler heuristics produce the largest intervals possible, represented by the largest g in each dimension.

The results of the symmetric scenarios AY×AY, GV×GV, and GM×GM are shown in Figures 6a, 6b, and 6c. They are symmetrical around the $X = Y$ plane, which is expected as $\{\tau_0, \tau_1\}$ achieves the same score as $\{\tau_1, \tau_0\}$ since both tasks are the same. With decreasing loop granularities g, the taskset response time R_{corun} generally increases, as the fixed-size scheduling cost (i.e., $S(|I_\tau|)$, see Equation 5) becomes more and more dominant. Note that to increase readability, we are not plotting configurations whose response time $R_{corun} > 2s$. However, in AY×AY and GV×GV local minima are exposed when the tiling granularity g_i of τ_i is a multiple of g_j of τ_j. As AY has a lower CCR, its intervals are shorter and more sensitive to this overhead, causing an offset in the minimum. In GM×GM, this effect is only a plateau, due to the higher CCR causing less memory contention.

For AY×AY and GV×GV, the maximum interval size compiler heuristics performs best also under co-scheduling. However, for GM×GM, the optimizer found a tiling granularity $g = 15$ that is more efficient than the $g = 20$ selected by the compiler heuristic (maximizing interval size). As data is reloaded at each PREM interval, the tile shape at $g = 15$ causes less inter-interval data reuse, leading to fewer reloads of data, and better performance.

Furthermore, the CCR impacts the OS, as memory-bound tasks require memory access for a larger portion of their execution time, which blocks their co-runner, thus increasing B^{memory}. The memory-bound AY achieves a maximum $OS = \frac{0.80}{1.12} = 0.71$, GV a maximum $OS = \frac{0.83}{0.92} = 0.90$, and the compute-bound GM a maximum $OS = \frac{0.71}{0.74} = 0.95$.

The results for the asymmetric scenarios AY×GV, AY×GM, and GV×GM are shown in Figures 6d, 6e, and 6f. These plots have a much more angular surface due to local optima where an even number of intervals of τ_0 can be executed during an interval of τ_1, or vice versa. In scenarios with GM, the best OS is always achieved when $g_{GM} = 15$ (as found before). For the co-running task however, we see two different effects for AY and GV. The GV×GM scenario is compute-bound enough to not introduce any significant memory blocking, and for GV the larger-is-better compiler heuristic leads to the best optimality score $OS = \frac{0.83}{0.90} = 0.92$. In the AY×GM scenario, however, R_{corun} can be reduced compared to the compiler selected $g_{AY} = 4096$ to $g_{AY} = 3369$. This implies a reduction by the tile size of $\sim 1/5$th, increasing the OS of AY×GM from $OS = \frac{0.80}{1.35} = 0.59$ to $OS = \frac{0.80}{0.93} = 0.86$, providing a 45% increase in the optimality score.

For AY×GV, reducing the tiling granularity g of both tasks yields the best result. Reducing the compiler selected $g_{AY} = 4096; g_{GV} = 11$ to $g_{AY} = 3830; g_{GV} = 8$ increases the optimality score from $OS = \frac{0.80}{1.12} = 0.71$ to $OS = \frac{0.80}{1.0} = 0.80$. This shows that reduced interval sizes can reduce the total response time R_{corun}, at the cost of task performance (due to increased $S(|I_\tau|)$), as illustrated in AY×GV: For the best version under co-scheduling, the per-task execution times

Table 2. Difference between the best and worst configurations of the Z vector, in percent of the response time R.

AY×AY	GV×GV	GM×GM	AY×GV	AY×GM	GV×GM
3.63%	5.93%	6.09%	3.98%	1.70%	2.80%

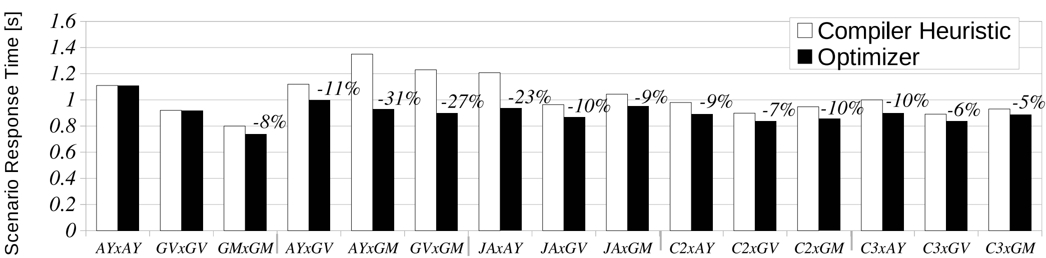


Fig. 7. The performance of previously proposed compiler heuristics compared to the best optimized solution.

were increased by 1% for AY and 4.2% for GV, due to finer-grained scheduling. However, this reduces the response time R_{corun} of the co-scheduled taskset by 11%.

5.2.1 Impact of Z overruling vectors. As the AY, GV, GM benchmarks are primarily loop based, the impact of the loop overruling vectors L are the most dominant. The impact of the combination overruling vector Z on the total execution time is presented in Table 2. While the impact is relatively small (the Z vector can only affect the execution time by a few percent), it might not be negligible in optimizing a system with tight deadlines D_{τ}.

Generally for three-phase PREM intervals, the behavior of loop-based and more sequential programs will be similar, as the L optimization vector can be expanded into the Z vector through unrolling, as outlined in Section 4.2. The impact these intervals have on the memory blocking time B^{memory} is limited by the size of the local memory $size(L)$, as all intervals, unrolled or not, must conform to the requirement of Equation 1. Therefore, these results are also representative for the impact of the Z vector in non-loop based programs.

However, the PREM model also supports single-phased *compatible intervals* [11] to execute portions of the code which can not be transformed by the compiler (e.g., *syscalls*) as a single memory phase. These intervals take ownership of the memory system during their entire execution (retaining the PREM *single-core equivalence* property), affecting B^{memory}. As both memory and computation time contribute to their $len(i)$, it is not limited by $size(L)$. The Z vector will start to dominate the L vector as the amount of *compatible intervals* with long execution times $len(i)$ increases. As *compatible intervals* should be avoided due to their bad behavior on B^{memory}, they are not otherwise covered in this paper.

5.2.2 Performance of Optimizer. To reach the described optima, the GA used the following parameters: Crossover rate $C = 12.5\%$, mutation rate $M = 12.5\%$, and population size of $N = 100$. The GA used a scaling window $W = 1$ and a pure selection strategy P. These parameters are chosen based on the discussion by Grefenstette [6]. We observe rapid convergence: on average 91% of the improvement was achieved already in the first epoch of the GA (worst being 88%). In this setup, 40 epochs take 6.5 hours on the TX2 on average. Within 10 epochs we achieved results within 3% of the best with only $N = 32$, which complete in under an hour. In larger PREM systems, scheduling more than two tasks will require more time to complete, and an optimization of the GA parameters and the workstation could be appropriate. However, we argue that the time to solution is tractable, as the optimizer is only invoked once. Furthermore, each

task within the system can be developed and tested with short iteration times, and only when all tasks are final, the proposed methodology is applied.

The presented GA-based *optimizer* uses only the response time R as optimization criteria. As shown, this is sufficient to solve the problem outlined in Section 3, and as it is a fundamental output parameter of any scheduler, it has the additional benefit of working with any PREM scheduler proposed in the literature. However, with tighter coupling of the *optimizer* to the scheduler, a more specialized *optimizer* could be possible. As part of our ongoing work we are exploring the explicit targeting of the intervals $i \in I_\tau$ that cause the maximum memory blocking time B^{memory} for other tasks $\bar{\tau}$ in the system. By focusing the optimization on this task τ it might be possible to reduce the time to solution.

5.3 Compiler vs Optimizer

Having determined that 10 epochs with $N = 32$ gives results close to the optima, we use these parameters to extend the evaluation to include three stencil kernels, *jacobi-1d* (JA), *convolution-2d* (C2) and *convolution-3d* (C3) (as shown in the bottom three rows of Table 1) from the PolyBench suite [13]. These benchmarks are executed in scenarios with the previously presented BLAS kernels, the results of which are shown in Figure 7. The first six set of bars refer to the BLAS scenarios described earlier.

The results show that the proposed optimizer can reduce the scenario response times up to 31% over the compiler generated intervals. Two lessons can be learned from this. First, once intervals are large enough to dominate the scheduling cost $S(|I_\tau|)$, it is more efficient to optimize for locality rather than maximizing interval size. This was shown clearly in the performance gain in the GM kernel in Figure 6c. As it only affects the task τ currently being compiled, is an optimization that could be implemented with compiler heuristics. Second, we validate that selecting the maximum PREM interval sizes can cause significant memory blocking B^{memory}, negatively impacting the response time R_τ as shown in in Equation 5. By selecting smaller interval sizes, the interleaving of memory and compute phases could be improved between the two tasks, as suggested by the motivating example in Figure 3, leading to a reduction in the response time R_{corun} in all but two scenarios.

As a rule of thumb, R_{corun} improvements are highest in memory bound scenarios (with large B^{memory}) where the CCR are sufficiently different between the tasks τ_0, τ_1 to allow effective interleaving. If both tasks are significantly compute bound, e.g., C3×GM, B^{memory} is low, and there is little to optimize. If both tasks are memory bound but the CCR similar, e.g., AY×AY, reducing the interval size still causes high B^{memory}. However, for memory bound scenarios where the CCR are not the same, e.g., AY×GM and JA×AY, scaling the interval sizes can lead to a large improvement due to better interleaving. These optimizations strictly depend on the interaction between tasks, and in contrast to the tiling optimization, there exist no compiler heuristics that could perform this optimization. Instead the solution can only be found with our proposed methodology.

6 RELATED WORK

A large amount of work has been done on PREM scheduling [1, 9, 18] and compilation [4, 9, 14]. All those works can be used together with the genetic optimizer presented in Section 4.3.1, as it only relies on the fundamental output parameter of the total response time R. A benefit of our proposed methodology is that it is not specific to one tool, which enables easy integration with custom or commercial tools.

Most closely related is the work by Soliman et al. [15], which also identifies the problem in Section 3. In contrast to our methodology, they integrate a segmentation-aware PREM scheduler into the compiler, which does not remove the

fundamental visibility problem of the translation unit, which this work addresses. Their scheduler could be adopted for use in our methodology to provide richer scheduler output to the optimizer. In contrast to their simulation-based evaluation, we execute real programs on real hardware and provide novel insights on the interplay between per-task and per-system performance from the PREM interval selection.

A non-PREM interference-aware scheduler was presented by Xiao et al [17]. Instead of fundamentally removing interference, it accounts for *instruction cache* interference in the WCET. However, this requires static analysis of the cache behavior, which is a difficult task on COTS systems.

Beside PREM, other software-based approaches to address memory interference in multi-core systems have been proposed [19, 20]. In contrast to the scheduling-based PREM, these are online methods that block memory accesses of interfering low-priority tasks to ensure that quality of service guarantees can be upheld for high-priority tasks. This gives PREM an advantage, as every PREM task in a system is guaranteed at design time to meet its real-time constraints.

Gupta et al [7] use busy-idle task graphs to defer task partitioning to scheduling time, similarly to overruling vectors in this work. However, these task-graphs are generated at compile time, which requires information about all tasks in the system. This makes their approach subject to the translation unit visibility problem addressed in this work. Furthermore, the underlying busy-idle profiles assume a loop-based message passing paradigm, and does not account for separate memory and compute phases over more generic programs.

7 CONCLUSION

In this work we have explored the impact and trade-offs between per-task and per-system performance in PREM systems. We have shown that due to memory serialization effects, selecting a performance-wise sub-optimal PREM interval configuration for tasks during compilation can improve the overall system response time. As these optimizations can not be implemented with compiler heuristics, we propose a novel methodology that is able to optimize at a system level, by taking the interactions between tasks into consideration. We have shown that our methodology can improve the response time of dual-core PREM execution tasksets by up to 31% without source code changes.

ACKNOWLEDGMENTS

This work has been partially funded by the European Union's Horizon 2020 project AMPERE (No. 871669), and ECSEL project COMP4DRONES (No. 826610).

REFERENCES

[1] A. Alhammad and R. Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *EMSOFT'14*, 2014.
[2] T. Back, U. Hammel, and H. . Schwefel. Evolutionary computation: comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1(1):3–17, April 1997.
[3] R. Cavicchioli, N. Capodieci, and M. Bertogna. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *ETFA'17*, 2017.
[4] B. Forsberg, L. Benini, and A. Marongiu. HePREM: Enabling predictable GPU execution on heterogeneous SoC. In *DATE'18*, 2018.
[5] B. Forsberg, L. Benini, and A. Marongiu. Taming data caches for predictable execution on gpu-based socs. In *DATE'19*, 2019.
[6] J. J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122–128, 1 1986.
[7] R. Gupta and M. Spezialetti. Busy-idle profiles and compact task graphs: compile-time support for interleaved and overlapped scheduling of real-time tasks. In *1994 Proceedings Real-Time Systems Symposium*, pages 86–96, 1994.
[8] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. 1977.
[9] J. Matejka, B. Forsberg, M. Sojka, P. Sucha, L. Benini, A. Marongiu, and Z. Hanzalek. Combining PREM compilation and static scheduling for high-performance and predictable MPSoC execution. *Parallel Computing*, 2019.
[10] NVIDIA. Jetson TX2. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/. Accessed: Nov 2019.

[11] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. In *RTAS'11*, 2011.

[12] R. Pellizzoni, A. Schranzhofer, Jian-Jia Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *DATE'10*, 2010.

[13] L.-N. Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*.

[14] M. R. Soliman and R. Pellizzoni. WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching. In *ECRTS'17*, 2017.

[15] M. R. Soliman and R. Pellizzoni. Prem-based optimal task segmentation under fixed priority scheduling. In *ECRTS'19*, 2019.

[16] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem: Overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[17] J. Xiao, S. Altmeyer, and A. Pimentel. Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 199–208, 2017.

[18] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Transactions on Computers*, 65(9):2739–2751, Sep. 2016.

[19] H. Yun, W. Ali, S. Gondi, and S. Biswas. BWLOCK: A dynamic memory access control framework for soft real-time applications on multicore platforms. *IEEE Transactions on Computers*, 66(7), 2017.

[20] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS'13*. IEEE, 2013.