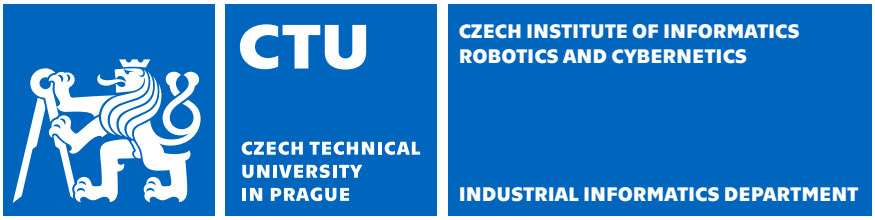



# Combining PREM Compilation and Static Scheduling for High-Performance and Predictable MPSoC Execution

Joel Matějka, Björn Forsberg, Michal Sojka, Přemysl Šůcha, Luca Benini, Andrea Marongiu, and Zdeněk Hanzálek

**DOI:** https://doi.org/10.1016/j.parco.2018.11.002
**Cite as:** J. Matějka, B. Forsberg, M. Sojka, P. Šůcha, L. Benini, A. Marongiu, and Z. Hanzálek. Combining PREM compilation and static scheduling for high-performance and predictable MPSoC execution. *Parallel Computing*, 2018

© 2018. This manuscript version is made available under the CC-BY-NC-ND 4.0 license, see http://creativecommons.org/licenses/by-nc-nd/4.0/

# Combining PREM Compilation and Static Scheduling for High-Performance and Predictable MPSoC Execution

Joel Matějka[a,c,*], Björn Forsberg[b], Michal Sojka[c], Přemysl Šůcha[c], Luca Benini[b], Andrea Marongiu[d], Zdeněk Hanzálek[c]

[a] *Czech Technical University in Prague, Faculty of Electrical Engineering, Technická 2, Prague, Czech Republic*
[b] *ETH Zürich, Institut für Integrierte Systeme, Gloriastrasse 35, Zürich, Switzerland*
[c] *Czech Technical University in Prague, Czech Institute of Informatics, Robotics and Cybernetics, Jugoslávských partyzánů 1580/3, Prague, Czech Republic*
[d] *University of Bologna, Viale Risorgimento 2, Bologna, Italy*

**Abstract**

Many applications require both high performance and predictable timing. High-performance can be provided by COTS Multi-Core System on Chips (MPSoC), however, as cores in these systems share main memory, they are susceptible to interference from each other, which is a problem for timing predictability. We achieve predictability on multi-cores by employing the predictable execution model (PREM), which splits execution into a sequence of memory and compute phases, and schedules these such that only a single core is executing a memory phase at a time.

We present a toolchain consisting of a compiler and a scheduling tool. Our compiler uses region and loop based analysis and performs tiling to transform application code into PREM-compliant binaries. In addition to enabling predictable execution, the compiler transformation optimizes accesses to the shared main memory. The scheduling tool uses a state-of-the-art heuristic algorithm and is able to schedule industrial-size instances. For smaller instances, we compare the results of the algorithm with optimal solutions found by solving an Integer Linear Programming model. Furthermore, we solve the problem of scheduling execution on multiple cores while preventing interference of memory phases.

We evaluate our toolchain on Advanced Driver Assistance System (ADAS) application workloads running on an NVIDIA Tegra X1 embedded system-on-chip (SoC). The results show that our approach maintains similar average performance to the original (unmodified) program code and execution, while reducing variance of completion times by a factor of 9 with the identified optimal solutions and by a factor of 5 with schedules generated by our heuristic scheduler.

*Keywords:* PREM, predictability, LLVM, static scheduling, Integer Linear Programming, NVIDIA TX1
*2010 MSC:* 00-01, 99-00

## 1. Introduction

Many real-time applications, such as autonomous cars and Advanced Driver Assistance Systems (ADAS), require both high computational performance and predictable timing. Although commercial-off-the-shelf (COTS) multi-core CPUs offer sufficient performance, it is difficult to predict task execution times because of cores competing for shared on-chip and off-chip resources such as main memory. The pessimism in worst-case execution times makes integration of complex systems with real-time requirements hardly feasible. In order to achieve the desired predictability, a predictable task execution model (PREM [1]) that guarantees *freedom from interference* can be employed.

*Corresponding author
*Email address:* Joel.Matejka@cvut.cz (Joel Matějka)

In PREM, application code is executed in sequences of non-preemptive *intervals* of two types: *predictable* or *compatible*. Predictable intervals are composed of memory *prefetch*, *compute* and memory *write-back* phases (in that order). The purpose of the *prefetch* phase is to load data needed in the *compute* phase to a core-local memory, such as L1 or L2 cache, to ensure that the compute phase does not compete for memory with other cores. *Compatible* intervals are those where the separation of memory and compute phases is not easily possible, which includes parts of the application as well as most system calls. The advantage of PREM is twofold: 1) memory phases have exclusive access to shared memory, limiting inter-core interference; 2) non-preemptive execution limits cache-related preemption delays [2].

In this paper, we achieve predictable execution with two steps: i) creation of PREM-compliant code (i.e., a sequence of predictable and/or compatible intervals); ii) scheduling intervals so as to guarantee mutually exclusive access to the main memory.

The creation of PREM-compliant code is a complex task, requiring knowledge of many low-level details. Such a task is better solved by optimizing compilers than by humans, particularly in the context of CPU codes, for which a large body of legacy code is involved. While previous research has discussed the desired features of such a compiler, and state-of-the-art analysis techniques for its practical design [1], the existing implementations still require the programmer to deal with low-level details.

In this work we present a compiler, based on the LLVM infrastructure [3], for the transformation of legacy CPU codes into PREM-compatible code. Specifically, this compiler performs several passes: i) identification of suitable portions of the code for conversion into predictable intervals; ii) splitting of the identified code into multiple predictable intervals, based on the size of available core-local memory; iii) generation of code for prefetch and write-back phases; iv) analysis of data dependencies between the intervals and their representation in the form of a directed acyclic graph (DAG), which is one of the inputs to a scheduling tool.

Scheduling memory phases on different cores to avoid mutual interference can be performed either on-line or off-line. On-line approaches are popular, because they do not require much *a priori* information, but their schedulability analysis (worst-case behavior analysis) is more challenging. Off-line scheduling is widely used in safety-critical systems. There, schedulability analysis is trivial, but schedule synthesis is difficult when the information needed for the synthesis is not known ahead of time. Fortunately, in many algorithms used in ADAS applications (e.g., FFT or matrix multiplication), it is known up front which operations need to be performed and which memory these operations access. For such algorithms, off-line scheduling approaches can easily find optimal schedules and provide high confidence in worst-case timing. One reason why people often prefer on-line approaches is that off-line scheduling leads to pessimistic results, because of pessimism in estimating worst-case execution time (WCET). This is, however, not the case with PREM, where the pessimism caused by unpredictable interference is limited, and thus, off-line scheduling can be practical and beneficial.

The main goal of this paper is to evaluate whether these expected benefits can be observed on real hardware with real-world algorithms. To achieve this goal, and as an additional important contribution of this paper, we present a fully integrated, complete implementation of PREM for a state-of-the-art embedded multi-core CPU. To the best of our knowledge, this is the first fully functional PREM implementation targeting COTS systems of this type.

In our previous work [4] we have presented a prototype compiler – capable of transforming regular loops into PREM-compliant code – coupled to a scheduling tool based on an ILP model and capable of optimally scheduling small task graphs. In this paper we significantly extend our previous work along several axes. First, the compiler can now not only analyze loops but also any other type of compute- and control flow-oriented parts of the program, both in parallel and sequential parts. It also improves the performance of the prefetching phases, thanks to the creation of optimized control flow that minimizes code footprint and improves cache behavior. Second, we overcome the poor scalability of the previous ILP scheduler with a new heuristic algorithm that extends state-of-the-art techniques [5] to handle schedules for hundreds of tasks in just a few seconds.

The schedule is computed from the information generated by the compiler (DAG) and from data obtained by simple single-core profiling of the generated code. We show that – compared to the ILP solver – the extended heuristic (i) solves nearly optimally (5–10% worse than optimum) small instances, with much

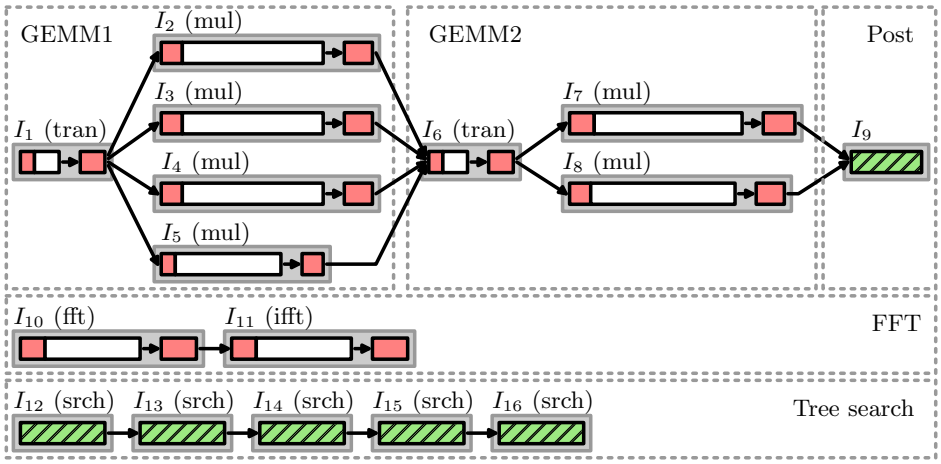



Figure 1: An example of PREMized ADAS scenario. Red rectangles are memory phases, white computations and hatched green are compatible intervals.

shorter solve times (over 45×) and (ii) enables solving much larger instances (hundreds of intervals as opposed to dozens) with reasonable solve times and (iii) single-core profiling is sufficient to execute the resulting application according to the generated schedule.

The paper is structured as follows. We introduce our system model in Section 2. In Section 3, we describe our compiler and its PREM-related passes. We follow with a description of our scheduling algorithms in Section 4. Section 5 and Section 6 describe implementation and evaluation both of the compiler and the scheduling on NVIDIA Tegra X1. Section 7 reviews related work and Section 8 concludes the paper.

## 2. System model

### *2.1. Target application template*

To demonstrate the approach presented in this paper, we selected a few algorithms widely used in autonomous driving systems. Our compiler flow transforms them into PREM-compatible code from which we create several execution scenarios. One such scenario is depicted in Figure 1. The first used algorithm is general matrix multiplication (GEMM), which is an essential operation in neural networks during forward propagation [6]. The second algorithm is fast Fourier transform (FFT) and inverse FFT (iFFT), which can be used in applications like visual object tracking, signal processing and similar. The third algorithm in our scenario is a memory intensive computation, typically encountered in binary search tree or graph traversing algorithms. Such algorithms are common-place in path planners, obstacle avoidance, and navigation.

The scenario in Figure 1 comprises two subsequent GEMMs, one FFT followed by inverse FFT and a sequence of binary tree searches. The first GEMM is formed by matrix transposition and four intervals of actual multiplication. The second GEMM works on smaller matrices – it has the transposition and only two multiplication intervals.

### *2.2. Target architecture*

The target platform of this work is the NVIDIA Tegra TX1 (Figure 2), a low-cost COTS system-on-chip (SoC) with four CPU cores. Each core is equipped with a core-local (non-shared) L1 cache memory, and a shared L2 cache. For hardware with shared cache memories, we assume that techniques such as cache coloring [7] are used to emulate cache partitioning. The TX1 employs random cache line replacement policy, which means that if there are no invalid cache lines available, a previously loaded cache line will be evicted at random to make place for new data. Thus, software mechanisms must be used to invalidate specific cache lines to ensure that loading of new cache lines does not randomly evict data that is still active (refer to Section 5.1 for more details). Each core is connected to a shared memory bus together with other devices such as the GPU. Solutions to handle interference from GPUs in heterogeneous SoCs such as the TX1 has

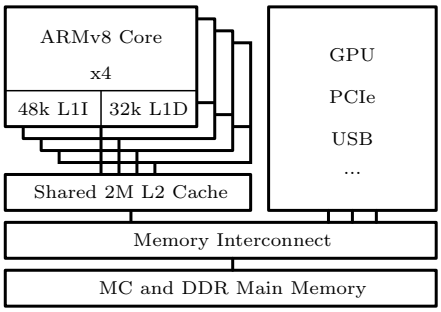



Figure 2: NVIDIA TX1 block diagram.

been previously proposed in the context of PREM [8]. The integration of our work with similar approaches is left for future work.

## 3. PREM compiler

To efficiently and predictably execute applications on the target platform, the compiler must produce code compliant with PREM. That is, code which is split into compatible and predictable intervals, with the latter ones composed of prefetch, compute and write-back phases. Our compiler, which can be seen as part of our toolchain in Figure 3, does this automatically, without the need for the programmer to specify additional pragmas/hints, beyond what he/she would use for program parallelization. Then, the compiler can generate a dependency graph of PREM intervals (similar to Fig. 1), which is used as an input for our scheduling tool described in Section 4.

We propose the design of a compiler based on the LLVM infrastructure that converts C/C++ code into PREM compliant code automatically. Figure 3 shows the block diagram of the PREM-related passes in the proposed compiler. The compiler can be separated into three parts: *Preprocessing*, *analysis*, and *transformation*. The preprocessing step performs standard transformations on the code to ensure that the code is in a known state before the main passes of the compiler execute. *Known state* in this context refers to canonicalization of the control flow graph to match certain patterns expected by the following passes. In addition to the built-in LLVM loop cannonicalization, the control flow graph is adjusted to include intermediate blocks to fully expose Regions that are single-entry single-exit, a property that is further described in the next subsection. Following this, the analysis passes (e.g., loop analysis) extract the required data from the source code, such as memory footprint information, upon which regions for PREM transformation can be selected. The main data that needs to be collected through analysis is the memory footprint which is used to select suitable PREM regions based on the available local memory. Once these have been selected, the transformation passes (e.g., outline) transform the program to conform to the requirements of PREM. In addition to returning the transformed program, the compiler also outputs the dependency graph of the PREM intervals, which dictate in which order the program must execute. This information is used by the scheduling tool presented in subsequent sections.

### *3.1. Preprocessing*

As outlined in the previous section, the first step in the PREM compiler is to detect all memory objects that have to be prefetched. Duplicate accesses to the same memory location through different pointers may lead to pessimistic results from the memory footprint analysis, as duplicate accesses are counted several times. To limit this pessimism, alias analysis is employed as part of the memory object detection, to identify duplicate accesses. Since the hardware caches will automatically handle any accesses to the same memory region, the program is guranteed to be correct even if duplicate memory accesses are not detected, but the size of PREM regions may be overestimated, leading to a larger amount of smaller intervals, that increase the scheduling complexity. Furthermore, as part of the preprocessing, the code is normalized into the

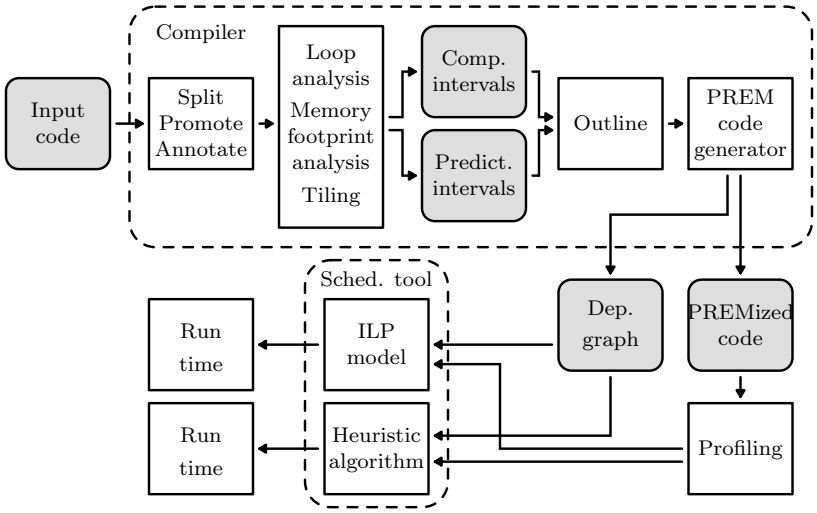



Figure 3: Block diagram of the toolchain proposed in this paper. Gray rounded boxes represent data, white rectangles are performed operations.

known state. Lastly, the basic blocks[1] that contain branches are split, such that the first block contains the computations, and the second block only contains the branch. As PREM intervals are created by one or more basic blocks, the splitting of branches into dedicated basic blocks enables the splitting of the pre-branch computation and the branch itself into separate PREM intervals. This later enables efficient creation of PREM intervals, by traversing the control flow graph. In addition to this, and for the same reason, basic blocks are split around call instructions, in a way similar to what has been previously proposed by Soliman et al [9].

### *3.2. Analysis*

The most important outcome of the analysis phase is to identify the portions of the code that are suitable for transformation into PREM intervals. There are two main requirements to such code portions: The first is that it should be possible to place prefetch and write-back operations such that data is ensured to be locally available at the point of use. We say that the prefetch point must *dominate* all points of use of the data, and the write-back point must *post-dominate* all points of use within the interval. The second requirement is that the data used between the prefetch and write-back points must fit into the local cache memory, as otherwise self-eviction would cause cache misses and violate the predictability guarantees that PREM strives to provide.

#### *3.2.1. Single-Entry Single-Exit Regions*

The LLVM infrastructure provides the concept of *regions*, which are defined as single-entry single-exit (SESE) portions of the code where the entry node dominates all other nodes in the regions, and the exit node post-dominates all other nodes in the region. Thus, this implicitly provides the first requirement of the PREM interval. A region node is defined as a single or a set of basic blocks. Regions are constructed within every function in the program, and function calls are used to link the regions of one function to another by creating a dedicated region for the call, as has been shown in [9]. Within a function, regions are represented as trees, where the root node is the region that covers the entire function. To conform to the SESE requirement, this means that every function may only return once, a limitation that is in line with the requirements set by the MISRA C [10] specification. For many programs the compiler is able to automatically restructure the code to have only a single return point, using built-in optimization passes.

[1]Basic blocks are sequences of instructions that execute without alteration of the control flow. A branching instruction is only allowed as a last instruction in the block.

Within each region, every further SESE region is represented as a region of its own, and added as a child to the region tree. By recursively identifying the SESE regions within every region, the tree is built top-down down to the smallest nodes. As each child region contains a subset of the nodes of the parent region, this leads to the following property:

***Property 1:*** *The memory footprint of a region $R$, is smaller or equal to that of its parent region $P$.*

Intuitively, if every memory access within the parent region $P$ occurs in a node that is also in $R$, the memory footprint of the two will be the same. However, if some of the memory accesses in $P$ are done in nodes that do not belong to $R$, the memory footprint of $R$ is smaller than that of $P$. This property becomes useful when PREM intervals are created in Section 3.2.3. By mapping all uses of memory objects to the region in which they are used, the memory footprint of each region can be calculated, as will be shown in Section 3.2.2.

Furthermore, the child regions are chosen as large as possible, i.e., the largest SESE regions within each region are used to populate the next level of the tree. This leads to another important property:

***Property 2:*** *A region node $R$ appears only once as a leaf node of the region tree, and every other direct or indirect parent of this leaf also references this node.*

As child regions are chosen as large as possible, they are non-overlapping, and any node in the control flow will only appear in one branch of the tree. To link the non-overlapping regions of a tree level together, the single successor of the single-entry single-exit region is encoded in the tree. Thus, it is possible to reconstruct the control flow graph from every layer of the tree. The first node is encoded as the entry point of the region.

### *3.2.2. Memory Footprint Analysis*

The memory footprint analysis is carried out in a bottom-up fashion on the region tree, starting from the leaf-nodes. For each region the individual memory objects accessed are identified, and their memory footprint calculated. While traversing the tree upwards, the contributions from every child region are merged, thus increasing the memory footprint as the traversal goes towards the tree root. The merge operation identifies any overlapping memory accesses between the child regions, to ensure that they are only counted once.

For sequential code the memory footprint can be calculated by simply summing up the contributions of each individual load or store operation. However, if the current region is part of a loop, extra steps must be taken to identify loop-variant memory accesses. In these cases, the memory object that is accessed depends on which iteration of the loop is executed. Consider for example a loop used to iterate over elements of an array, in this case a different array element is accessed during each loop iteration, and the memory footprint depends on the number of iterations executed. Loop-variant accesses are characterized by their dependence on the loop *induction variable*, i.e., the variable that changes during each iteration. To understand which memory objects that are accessed, every value that the induction variable takes during the execution of the loop must be calculated. All modern compiler infrastructures provide support for induction variable-based analysis of loop expressions, such as the polyhedral model or the scalar evolution analysis [11]. We rely on the latter to implement our techniques.

### *3.2.3. PREM Interval selection*

Once the memory footprint of every region is calculated, it is possible to traverse the region tree to select PREM intervals whose memory footprint is small enough to fit in the local memory. This is a recursive and greedy process that tries to select the largest regions possible. If a region is too large to fit into the local memory, the recursion continues to the children of the region. As the memory footprint decreases as the recursion continues, for each recursion step the memory footprint of the nodes becomes smaller, and by selecting them as separate PREM intervals, a large program can be efficiently divided into smaller chunks. Due to the non-overlapping property of the region tree, it is guaranteed that each part of the program belongs to a unique PREM interval.

There are two types of regions that require special handling, *branch regions*, and *loop regions*, which we differentiate from *common regions*. *Branch regions* are identified by their entry point containing a node with two or more successors, but not belonging to the condition test of a loop. The simplest example of this is an if-statement. As the regions are SESE, a region starting with a branch node will by definition

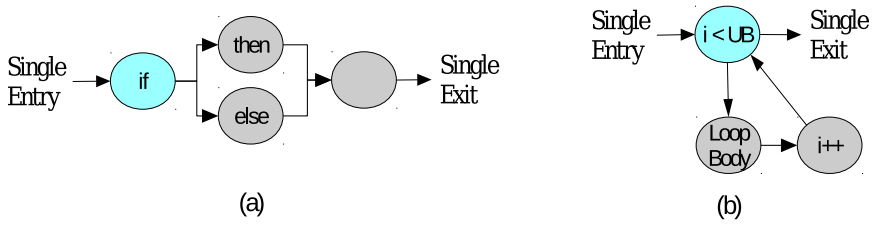



Figure 4: The control flow of a branch (a) and a loop (b). Note that the SESE property of the loop, while not immediately clear, can be identified by imagining an empty node preceeding and postceeding the loop condition (colored).

cover the entire branch until the control flow rejoins, as shown in Figure 4a. For this reason we call the first node of a *branch region* the *branch fork* and the last node the *branch join*. If the entire *branch region* fits into local memory, it can be selected as a PREM interval as usual. However, if the recursion continues into the block, separate PREM intervals need to be created for each branch outcome, as the preceeding PREM interval could otherwise end in either of the branch outcome regions. In such a case, the longer of the two execution paths would dictate the WCET of the interval, furthermore, a larger amount of data may need to be prefetched, but only partially used, once again increasing the WCET of the PREM memory phase. For this reason, if a full branch region does not fit into the local memory, an individual PREM interval is created for each branch outcome.

*Loop regions* are identified by their entry point being the condition test for a loop, thus containing two successors: one to enter/continue the execution of the loop, and one to exit the loop if the end condition is fulfilled, as shown in Figure 4b. If a full loop region fits in the local memory it can be selected within a PREM interval, however if it does not then two steps are necessary. First, as the loop condition test is a branch, a new PREM interval must be created in the same manner as for *branch regions*, and second, the loop must be split into multiple smaller chunks which fit in the local memory. The process of splitting is achieved through *tiling* [11], in which the iteration space is divided into multiple smaller parts (see groups of "mul" intervals in Figure 1). Thus, the iteration space of the original loop is divided into multiple smaller parts. By using the memory footprint information from scalar evolution, it is possible to compute how the memory footprint changes as the iteration space is reduced. Once the *tiling factor*, i.e., how many of the original loop iterations to execute during each tile, is computed, the tiled loop can be selected as a PREM interval, as it now fits in local memory.

Based on the three above cases, *common regions*, *branch regions*, and *loop regions*, the recusive PREM interval selection algorithm is presented in Algorithm 1. The long $if$-statement tries to select the current region node to a PREM interval, where lines $2-4$ try to include the node in the current interval. If the current node does not fit into the current interval, different measures are taken depending on the node type. Lines $5-9$ handle the creation of separate PREM intervals for branch outcomes, lines $10-14$ handle the tiling and reselection of loops, and lines $15-16$ handle the recursion through the tree for regular nodes. If the node does not fit into the current interval, and is a leaf node of the tree, lines $17-21$ create a new PREM Interval and tries to reselect the node when all local memory is available. Once the full current region is selected, or it has been selected through division into smaller parts by the recursive step, the last line continues the PREM interval selection to the successor node. Thanks to this, the PREM regions are guaranteed to be selected as sequential nodes of the control flow. At the end of the function, each node will be part of exactly one PREM interval, either directly, or through a direct or indirect parent node. The splitting of basic blocks around branches, as performed during the preprocessing stage, maximizes the chance that branches can be selected as a single PREM interval, without having mutually exclusive PREM intervals per branch outcome, which will later simplify the scheduling step. This effect comes from the accesses that are not dependent on the branch outcome are separated from the accesses within the branch, and thus do not effect the memory footprint of the branch region.

```
1: Input: currentRegion
2: if currentRegion.footprint ≤ currentInterval.availableMem then
3:    currentInterval.addRegion(currentRegion)
4:    currentInterval.availableMem− = currentRegion.footprint
5: else if currentRegion.isBranch then
6:    for all childRegion in currentRegion.childRegions do
7:       currentInterval = newPREMInterval(fullCacheSize)
8:       Recurse on childRegion
9:    end for
10: else if currentRegion.isLoop then
11:    currentInterval = newPREMInterval(fullCacheSize)
12:    currentRegion′ = tile(currentRegion, fullCacheSize)
13:    currentInterval.addRegion(currentRegion′)
14:    currentInterval = newPREMInterval(fullCacheSize)
15: else if currentRegion.hasChildren then
16:    Recurse on currentRegion.entryPoint
17: else
18:    currentInterval = newPREMInterval(fullCacheSize)
19:    currentInterval.addRegion(currentRegion) or Fail
20:    currentInterval.availableMem− = currentRegion.footprint
21: end if
22: Recurse on currentRegion.successor
```

**Algorithm 1:** Pseudo-code for the PREM interval selection, adding Region nodes from the Region tree to a PREM Interval.

### *3.3. Transformation*

Once PREM intervals have been selected by the recursive selection algorithm, the transformation takes place. First, all region nodes belonging to a PREM interval are outlined into a new function, and in connection to each call to an outlined function, calls to two new functions are inserted. These functions will implement the PREM *Prefetch* and *Writeback phases* for the outlined function.

Based on the accesses identified as part of the analysis phase, the *Prefetch* and *Writeback phases* are created from scratch, by generating the minimal amount of code that is required to prefetch or evict the data touched by each PREM interval. For the *Prefetch phase*, this is done by creating a loop that executes the *prfm pldl2keep* or *prfm pstl2keep* prefetch instructions, for loads and stores respectively. For the Writeback phase, the prefetch instructions are replaced with *dc civac* instructions, which cleans and invalidates an address from the cache. Data is moved to and from the cache at the granularity of a *cache line*, which consists of a fixed number of sequential bytes, typically between 32 and 128. To optimize the execution of the *Prefetch* and *Writeback phases*, the compiler is able to determine sequential access patterns, and increases the stride of the loop to only touch each cache line *once*. This means that the number of prefetch or flush instructions is reduced by a factor of $\frac{cachelinesize}{elementsize}$, while still moving all the data to the cache. As an example, a sequential access pattern of *floats* (4 bytes), on a system with 64 byte cache lines, reduces the amount of prefetch/evict instructions by a factor of 16. The effects of this on a real system is further detailed in Section 6.

The compute function is kept as-is, but is now ensured to hit in the cache on every access, assuming that the cache replacement policy did not self-evict any of the prefetched addresses. We experimentally evaluate these effects as well, in Section 6. Methods to prevent cache conflict misses are left for future work.

Once all the transformations have been applied, a dependency graph which specifies the correct program order of the PREM intervals is produced, such that this property can be respected by the scheduler.

#### *3.3.1. Dependency Graph Generation*

In addition to performing the transformations, at the end of the Analysis phase a directed dependency graph is implicitly created from the PREM intervals that have been selected. Due to the selection process, each PREM interval, except for the program entry and exit, has predecessor and successor intervals implicitly defined through the control flow graph of the task. To schedule these PREM intervals, the scheduler requires a directed ayclic graph (DAG), however, the implicitly generated graph would contain cycles if loops have

been tiled into several PREM intervals. For this reason, before the dependency graph is passed to the scheduler, these loops are unrolled, at a tile basis, to remove the cycles. Note that, even if the loops would not be fully unrolled within the code, for code size reasons, the dependency graph is always fully unrolled. This process is required for the scheduler to work correctly, and is required even if the amount of unrolled intervals is very large. However, compared to loop unrolling, the unrolling on a tile basis produces much fewer nodes in the graph, and does not include instruction level information, but only interval identification information. Through this process, the possibly cyclic graph has been turned into a non-cyclic dependency graph, which is forwarded to the scheduler.

## 4. Scheduling

After the code and dependency graph have been generated by the compiler, we use the scheduling algorithm described in this section to schedule parallel execution of the code on the multi-core target platform. The goal is to minimize completion time ($C_{\mathrm{MAX}}$) of the last executed interval, while simultaneously ensuring no interference at memory bus. We first describe the PREM application model (Sec. 4.1) and the scheduling model (Sec. 4.2). Then, we briefly summarize the ILP model from our previous work [4] (Sec. 4.3) and finally introduce the new heuristic (Sec. 4.4). A comparison between optimal and heuristic solution is presented in Section 6.

### *4.1. PREM application model*

The application transformed into PREM compliant code has the following structure: It is a static set of PREM intervals $\{I_1, I_2, \ldots\}$ with dependency relations in the form of a directed acyclic graph (DAG). An example of an application execution scenario transformed into PREM compliant code is shown in Figure 1. Intervals $I_9$ and $I_{12}$–$I_{16}$ are compatible intervals and the rest consists of predictable intervals. Red rectangles represent *prefetch* and *write-back phases*, white rectangles are *compute phases*. For the sake of model simplicity, we consider compatible intervals requiring exclusive memory access (equivalent to predictable interval with zero-length *compute* and *writeback phase*).

For each phase of our model, we need to know its WCET. Compared to unrestricted execution models, determining the WCET of PREM compliant code is pretty straightforward as the PREM model limits possible inter-core interference by ensuring (i) exclusive access to the shared memory in prefetch, write-back and compatible phases and (ii) availability of all required data in local memory in compute phases. As demonstrated in our experimental evaluation, WCET times obtained by a simple measurement-based method (profiling) match real execution times with sufficient accuracy. More complex and conservative WCET estimation techniques, based on static methods [12] can be used, as the presented methodology is agnostic to how the bounds were retrieved.

### *4.2. Scheduling Model*

To construct an application schedule, we execute our scheduling algorithm on the scheduling model graph derived from the PREM application model described above. Here, we present only an informal description of the scheduling model required for understanding the heuristic algorithm. For the formal description of the model, we refer an interested reader to [4].

We formulate the scheduling problem as a resource-constrained project scheduling problem (RCPSP) with multi-resource activities. We use a trivial example in Figure 5 to illustrate the conversion of a set of PREM intervals into our scheduling model consisting of $n+2$ *activities* $\mathcal{V} = \{0, 1, 2, ..., n+1\}$. Activities 0 and $n+1$ (not shown in our trivial example) denote "dummy" activities which represent the project beginning and the project termination, respectively. We also show how the final schedule is constructed from the RCPSP solution. The example consists of two predictable intervals ($I_1$ and $I_2$) composed of prefetch (P), compute (C) and write-back (W) phases, with known execution times $p_i$ . The intervals need two resources to execute: CPU core with core-local memory and shared memory controller (MC). CPU core is required by all phases, MC only by prefetch and write-back phases.

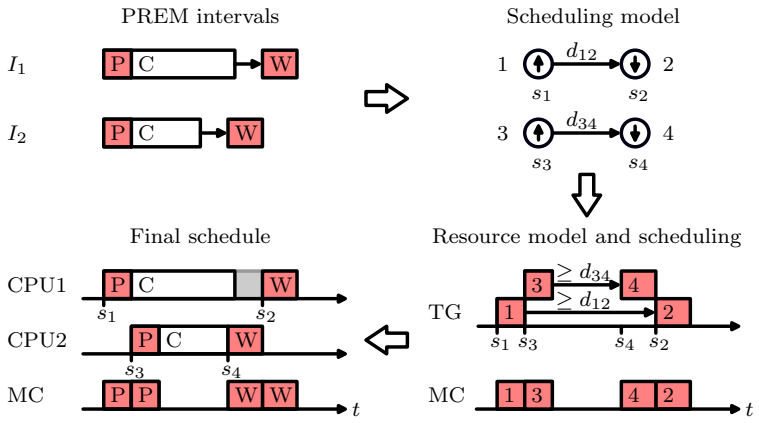



Figure 5: Translation of two PREM intervals into scheduling model and backwards

Since PREM intervals are non-preemptive (another interval cannot be scheduled between start of prefetch and end of write-back phase) and compute phases do not require any additional resource, we omit the compute phase in our scheduling model, and create two activities representing prefetch and write-back phases and a temporal constraint between start times of these activities representing the total length of the prefetch and compute phases. In our example, interval $I_1$ is converted into two activities 1 and 2 with start times $s_i$ (calculated by the scheduling algorithm) and temporal constraint $d_{12}$.

Combination of non-preemptive intervals and symmetric multiprocessor system enables modeling of all CPU cores as one so-called *take-give resource* (TG) [5]. The take-give resource may be seen as a counting semaphore with the capacity $Q$ equal to the number of available CPU cores. In contrast to scheduling with classical resources, TG resources do not require the total occupation time (i.e., time between take and give operations) to be known in advance. The up-/down-pointing arrows in circles represent the take/give operation, which take/give one unit of the TG resource with total capacity of 2.

Our scheduling algorithm takes the activities as input and produces the schedule in which all resource requirements and temporal constraints are met. As our target platform is a symmetric multiprocessor system, it is not necessary to assign activities to particular cores. It is sufficient to determine start times of all prefetch and write-back phases (or their order). Non-preemptivity of intervals ensures that the intervals do not migrate to other cores. Obtained start times are propagated back into the PREM model, and the run-time scheduler dispatches the intervals to a random free core at corresponding times. Notice that the write-back phase does not need to start immediately after the compute phase. A delay may occur when the memory controller is occupied by an activity executed on another core (e.g., write-back phase of $I_1$ waits for completion of the write-back phase of $I_2$ in Figure 5).

We model compatible intervals similarly to predictable intervals with zero length of compute and write-back phases. Therefore, a compatible interval creates two activities where the length of the first and the value of the linking temporal constraint is equal to the length of the interval and the second activity has zero length.

Figure 6 shows a scheduling model for a large-scale scenario with 111 intervals (222 activities). The scenario is further detailed in experimental evaluation Section 6.

### *4.3. ILP formulation*

The ILP formulation of the scheduling problem is shown in Figure 7. The inequalities (2) represent temporal constraints between the activities, inequalities (3) and (4) express occupation of the memory controller and inequalities (5)–(10) stand for the take-give resource constraints. Occupation time of a take-give resource by an activity is represented by variable $\tilde{p}_i$. Each take-give resource occupation is indicated by binary variable $a_{i,j}$ and take-give resource assignment by $\tilde{z}_{i,j}$. Binary variables $x_{i,j}$, $\tilde{x}_{i,j}$ and $\tilde{y}_{i,j}$ reduce corresponding constraints. Finally, the objective function of the ILP model (1) minimizes the start time

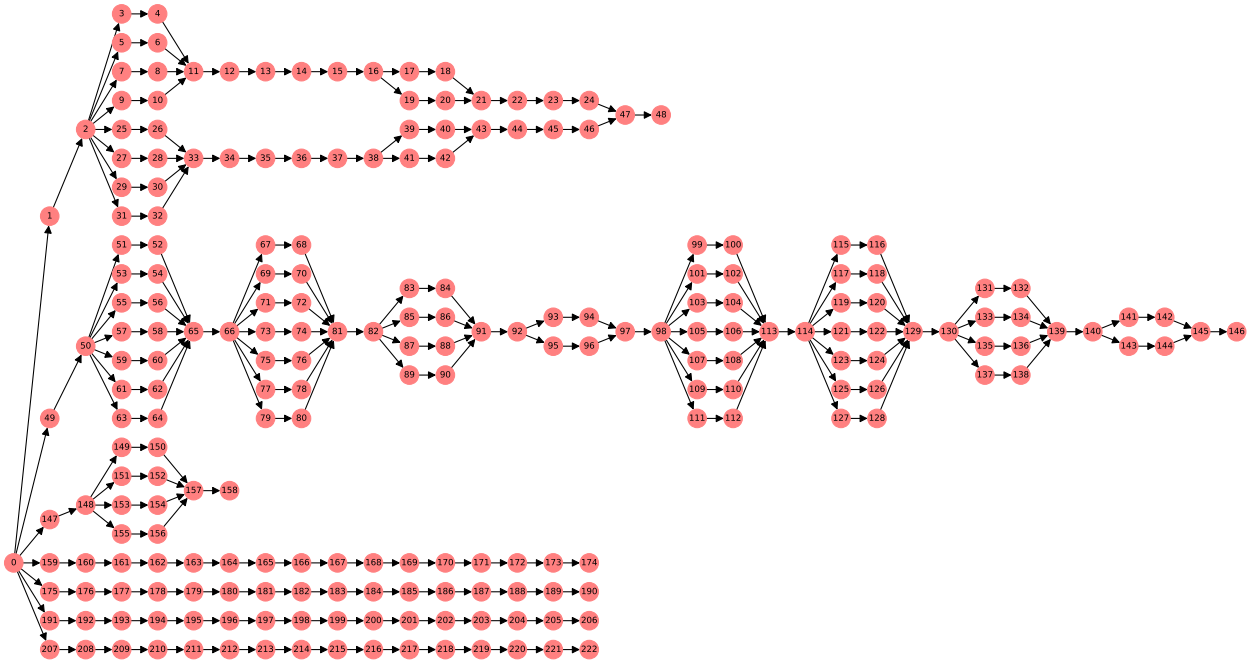


Figure 6: Scheduling model DAG constructed for scenario 5

of the dummy activity $n + 1$, i.e., the last activity of the schedule. More detailed description of the ILP formulation is given in [4].

### *4.4. Efficient heuristic*

The ILP formulation of the scheduling problem, proposed in [4], is capable of solving only small instances in a reasonable time. To create efficient schedules for large-scale instances, we extended a state-of-the-art heuristic algorithm, originially proposed by Hanzalek [5]. The core of the heuristic is a priority-based list scheduling algorithm with unscheduling step, which can remove already scheduled tasks conflicting with the task currently being scheduled. To achieve near-optimal solutions on large-scale instances, this basic algorithm uses a few additional techniques that improve the results. One such a technique is the time symmetry mapping, which allows construction of schedules in both forward and backward time orientation. Others are propagation of information about conflicting tasks into subsequent iterations, and parallelization of the algorithm that enables reduction of solving time. Our main contribution in this work is the extension of the algorithm to support multi-capacity take-give resources, as described in Section 4.2.

#### *4.4.1. Algorithm overview*

The proposed heuristic is illustrated in Algorithm 2. After initialization (lines 1–6), a bounded amount of while loop iterations (from line 7 on) is performed. The goal of the loop is to iteratively tighten the bounds maintained by the algorithm. In each iteration a decision problem "does a solution with these bounds exist?" is solved and new instances for next iterations are created. The best solution found during the iterations is the output of the heuristic. The algorithm consists of the following building blocks:

*Initialization.* During the initialization phase (lines 1–6), the algorithm precalculates maximal distances $d_{i,j}$ between all activities in a graph and sets initial lower bound $LB = d_{0,n+1}$ and upper bound $UB = \sum_{i \in \mathcal{V}} p_i$. First item is enqueued into the solution queue.

**input :** an instance $I$
**output:** best schedule $S^{best}$

Calculate $d_{i,j}$ $\forall (i,j) \in \mathcal{V}^2$;
Calculate $LB$ and $UB$; $C = (LB + UB)/2$;
$S^{best} = \emptyset$; $priority(i) = d_{i,n+1} \quad \forall i \in \mathcal{V}$;
$failureCounter = 0$; $priorityHashSet = \emptyset$;
$I^{forward} = I$; $I^{backward} =$ `TimeSymmetryMapping` $(I)$;
$schedulingParameters$.Enqueue($\left\{ I^{forward}, C, priority \right\}$);
**while** $LB < UB$ *and* $|schedulingParameters| > 0$ **do** `// the stopping condition`
    $\{I^c, C, priority\} = schedulingParameters$.Dequeue();
    $hash =$ `hash` $(priority)$;
    **if** $hash \in priorityHashSet$ *and* $C > UB$ **then**
        **continue**; `// this tuple was already processed`
    **end**
    $priorityHashSet$.Add$(hash)$;
    $S =$ `findSchedule`$(I^c, C, priority)$;
    **if** $S$ *is feasible* **then**
        **if** $UB > C_{max}(S)$ **then** `// a new better solution was found`
            $UB = C_{max}(S)$;
            $LB = \min(LB, \lceil 0.8 \cdot UB \rceil)$;
            $S^{best} = S$;
        **end**
        $C^{new} = UB - 1$;
        $priority = (C - s_i)_{i \in \mathcal{V}}$; `// get new priority from the start time`
    **else**
        $failureCounter++$;
        **if** $failureCounter \geq \log_2(n)$ **then**
            $LB = LB + (UB - LB)/4$;
            $failureCounter = 0$;
        **end**
        $C^{new} = \min(UB, \lceil 1.1 \cdot C \rceil)$;
    **end**
    $priority^1 = (C - priority_i)_{i \in \mathcal{V}}$; `// reverse priority`
    $priority^2 =$ `modifyPriority` $(priority, S)$; `// modify original priority`
    **if** $I^c == I^{forward}$ **then** `// change orientation of the instance`
        $I^{c1} = I^{backward}$;
    **else**
        $I^{c1} = I^{forward}$;
    **end**
    $I^{c2} = I^c$;
    $schedulingParameters$.Enqueue($\{I^{c1}, C^{new}, priority^1\}$);
    $schedulingParameters$.Enqueue($\{I^{c2}, C^{new}, priority^2\}$);
**end**

**Algorithm 2:** Iterative Resource Scheduling algorithm

$$\min s_{n+1} \tag{1}$$

subject to:

$$s_j - s_i \geq d_{ij}, \qquad \forall (i,j) \in \mathcal{V}^2 : \ i \neq j \tag{2}$$

$$s_i - s_j + UB \cdot x_{ij} \geq p_j, \qquad \forall (i,j) \in \mathcal{V}^2 : \ i \neq j \tag{3}$$

$$s_i - s_j + UB \cdot x_{ij} \leq UB - p_i, \qquad \forall (i,j) \in \mathcal{V}^2 : \ i \neq j \tag{4}$$

$$\tilde{p}_i = s_l + p_l - s_i, \qquad \forall (i,l) \in \mathcal{V}^2 : \ a_{il} = 1 \tag{5}$$

$$s_i - s_j + UB \cdot \tilde{x}_{ij} + UB \cdot \tilde{y}_{ij} \geq \tilde{p}_j, \qquad \forall (i,j) \in \mathcal{V}^2 : \ i \neq j \tag{6}$$

$$s_i - s_j + UB \cdot \tilde{x}_{ij} - UB \cdot \tilde{y}_{ij} \leq UB - \tilde{p}_i, \qquad \forall (i,j) \in \mathcal{V}^2 : \ i \neq j \tag{7}$$

$$-\tilde{x}_{ij} + \tilde{y}_{ij} \leq 0, \qquad \forall (i,j) \in \mathcal{V}^2 : \ i \neq j \tag{8}$$

$$\tilde{z}_{iv} + \tilde{z}_{jv} - 1 \leq 1 - \tilde{y}_{ij}, \qquad \forall (i,j,l,h) \in \mathcal{V}^4, \forall v \in \{1,...,Q\} : \ i \neq j, \ a_{il} \cdot a_{jh} = 1 \tag{9}$$

$$\sum_{v=1}^{Q} \tilde{z}_{iv} = a_{il}, \qquad \forall (i,l) \in \mathcal{V}^2 : \ a_{il} = 1 \tag{10}$$

the domains of the input parameters are: $d_{ij} \in \mathbb{R}_0^+, \ p_i, UB \in \mathbb{R}_0^+, \ a_{il} \in \{0,1\}$
the domains of the output variables are: $s_i \in [0, UB - p_i], \ \tilde{z}_{iv} \in \{0,1\}$
the domains of the internal variables are: $\tilde{p}_i \in [0, UB], \ x_{ij}, \tilde{x}_{ij}, \tilde{y}_{ij} \in \{0,1\}$

Figure 7: ILP formulation of the problem

*Solution queue.* Input to each iteration is a tuple $(I^C, C, priority)$, where $I^C$ is the current instance, $C$ is the requested maximal schedule length and *priority* is a vector of priorities of length $n+2$. These tuples are stored in a queue denoted as $schedulingParameters$. Although the algorithm could be formulated in a more compact and elegant way using a recursion, we use the queue because it enables easy parallelization of the algorithm. The queue-based formulation also allows the possibility that multiple tuples in queue result in the same solution after several iterations. To avoid solving of redundant instances, the algorithm computes and stores a hash of the priority vector in a hash table, and skips future queue entries with the same hash.

*Find schedule.* The core of the algorithm is the function *findSchedule* called at line 14 and shown in Algorithm 3. It tries to solve the decision problem mentioned in the overview by transforming the vector *priority* into a schedule $S$ that has its completion time ($C_{\max}(S)$) smaller than $C$.

The function iterates until a feasible schedule is found or the *budget*, which depends on the number of activities, is depleted. In each iteration, an activity with the highest priority which is not scheduled yet is chosen and the earliest possible start time $ES_i$ and the latest possible start time $LS_i$ of the activity $i$ is calculated. Then the *findTimeSlot* function tries to fit the activity into the current schedule at an earliest possible time slot with respect to already scheduled activities and dependencies. If a free slot is found, the start time $s_i$ of the activity is set so that the activity fits into the schedule, and the activity is added into the set of scheduled tasks (no conflicting activity exists). Otherwise, the task is forced into schedule with start time $s_i$ and all conflicting activities (tasks that potentially block the insertion of the current task into the schedule) are unscheduled. When the finding of the free time slot is unsuccessful for the first time, the start time is set to $ES_i$, otherwise $s_i = s_i^{prev} + 1$ where $s_i^{prev}$ is the previous start time.

*Schedule evaluation.* If the feasible schedule is found, new bounds based on $C_{max}(S)$ and new priorities are calculated. The $C^{new}$ is decreased to $UB - 1$ and when the schedule is the best solution so far, it is stored in $S^{best}$, the $UB$ is updated to $C_{max}$ and $LB$ is decreased to $\min(LB, \lceil 0.8 \cdot UB \rceil)$ which gives the algorithm more time to find a better solution. In the second case, the algorithm increases $C^{new}$ to $min(\lceil 1.1 \cdot UB \rceil, \lceil 1.1 \cdot C \rceil)$ in order to allow the escaping from the current local optimum. The algorithm also

```
findSchedule(I, C, priority)
s_i = −∞ ∀i ∈ V;
scheduled = ∅;
budget = budgetRatio · n;
while budget > 0 and |scheduled| < n + 2 do
    i = arg max_{∀i∈V:i∉scheduled} (priority_i);
    ES_i = max_{∀j∈V:j∈scheduled} (s_j + d_ji);
    LS_i = C − p_i;
    {slotFound, s_i} = findTimeSlot(i, ES_i, LS_i);
    if !slotFound then
        s_i = s_i^prev + 1;
    end
    unscheduled = unscheduleConflictingActivities (i, s_i);
    scheduled = scheduled \ unscheduled;
    scheduleActivity(i, s_i);
    scheduled = scheduled ∪ {i};
    budget = budget − 1;
end
return (s_i)_{i∈V};
```

**Algorithm 3:** Priority-rule based function with an unscheduling step

counts unsuccessful iterations and when the *failureCounter* exceeds a given threshold, the $LB$ is increased to $LB + (UB - LB)/4$ in order to fulfill the stopping condition after limited amount of unsuccessful iterations.

The *priority* vector is changed in every algorithm iteration and it progressively converges to the priorities, which allow to find a better solution in latter iterations. If the schedule $S$ is feasible, the priorities are updated according to the start times of activities in the schedule – the latter start time, the lower priority.

It does not matter whether the *findSchedule* function finds a feasible schedule or not. In both cases two new tuples are generated and inserted into the queue. The first tuple contains always the time symmetric instance and appropriately reversed vector of priorities. In the second tuple is an instance with preserved orientation and modified priorities with respect to the most frequently conflicting activity during the previous schedule construction. During the schedule creation, the *findSchedule* function registers conflicts between activities, which are subsequently evaluated. For the activity couple with the maximum number of conflicts, the priorities are swapped and higher priority is propagated backward into priorities of previous activities so that while previous activities have lower priority, the priorities are increased by a difference between the swapped priorities.

*Time symmetry mapping.* Time symmetry mapping transforms input instance so that activities are scheduled in reverse time orientation (from activity $n + 1$ to 0). The process of conversion is explained more in detail in [5].

An example of a feasible schedule found by our heuristic algorithm is depicted in Figure 12. The schedule corresponds to scenario 8×1 (scenario in Figure 1 duplicated 8 times in sequence), which is more detailed in Section 6.4.

## 5. Implementation

We use the LLVM compiler infrastructure [3] for source code analysis and PREM compliant code generation. The passes are designed such that they offer modularity and are as independent as possible, and information is passed between the passes using *ad-hoc* metadata. The presented technique poses the following two restrictions to the supported C/C++ codes: The code cannot contain any form of recursion, and all loops have to be bounded by a constant or statically computable value to enable scalar evolution analysis. It has to be stressed that these restrictions are in line with the requirements of typical coding standards adopted in the automotive domain, such as the MISRA guidelines [10]. In light of this, these restrictions do not impose any severe limitations to real applications in the target domain.

### 5.1. Limitations in the current setup

In addition to the previously listed limitations on the supported codes due to the technique itself, the current implementation has some further limitations.

Currently, the compiler does not detect and prefetch stack variables (e.g., spilled registers and function arguments), which implies that accesses to stack data may still cause cache misses during the compute phase. The size of the stack is not known until the end of the compilation, and instrumenting them from within the compiler would require special instrumentation in the compiler backend. It may instead be better to prefetch the stack at runtime, using known techniques previously presented as part of LightPREM [13]. However, these accesses only make up a small portion of the total memory accesses of the program, and their impact on the predictability is thus low, as we show in the empirical evaluation in the next section.

Even when data is prefetched, the target platform does not guarantee that the data will still be available in the cache at the start of the compute phase, due to the random cache replacement policy employed. In caches with random replacement policy, the cache controller randomly selects a candidate cache line and evicts it to make space for new data when necessary. This strategy breaks the PREM model because we can not deterministically select which data will stay in the cache. However, in experiments below, we show that also random cache replacement policy can be partially deterministic. When the new data is transferred into the L2 cache, the controller fills invalidated cache lines first. It is therefore possible to minimize the risk of evicting active data by ensuring that cache lines that are no longer needed are explicitly evicted from the cache. Therefore we flush and invalidate the entire cache at the beginning of the schedule, and subsequently, we flush and invalidate every cache line used during the execution of PREM intervals in the respective write-back and compatible phases. Because of the need to flush every cache line used during computation, data that are shared between cores are duplicated to ensure that a write-back phase on one core does not affect any other cores.

The L2 cache to which the prefetches are done is shared between all cores, which means that, even though the data itself is duplicated, data accesses of the different cores may still evict each others data if they map to the same index in the cache. Solutions to this problem have been proposed in the literature, e.g., through the use of cache coloring [7], which ensures that only a single core will access each portion of the shared cache. Currently, such mechanisms have not been implemented on the target platform, and the compiler treats the L2 as a private memory. In order to minimize possible evictions due to accesses from multiple cores, our compiler only allocates part of the actual L2 cache capacity. We observe a significant increase of cache misses during compute phases when the allocated capacity is larger than three-quarters of the actual capacity. Therefore we selected only half of the actual capacity.

A resulting effect of the need for data eviction at the end of each phase, is that data sharing between two intervals running in parallel becomes problematic. The reason for this is that one task may evict data that is still used within another task on another core. For this reason, the current setup requires the adding of a synthetic dependency between two tasks that share data, to ensure that they will not be scheduled at the same time. In doing this, it is guaranteed that one task will not evict the data of another. To support data sharing between parallel tasks on systems that require explicit data evictions, the eviction of data structures could be made conditional on a control bitmask, generated by the schedule. This bitmask would encode which task is responsible for prefetching shared data, and which task is responsible for evicting this. Once the scheduler has produced the final schedule, this information could be injected into the system. This would require an extension to the scheduler, and is out of scope for this work.

It has to be underlined that the mentioned limitations can be addressed through known techniques, and do not constitute an inherent limitation of the presented methodology. With the exception of the stack data limitation, the presented limitations are also specific to the random replacement cache policy, and may not be present in all COTS hardware. Moreover, the compiler works at the intermediate representation level and hence is portable to other architectures such as the Intel x86.

## 6. Experimental evaluation

The evaluation section begins with an evaluation of the compiler optimizations, and an exploration of the predictability and performance characteristics of the PREM kernels. Following this, in order to

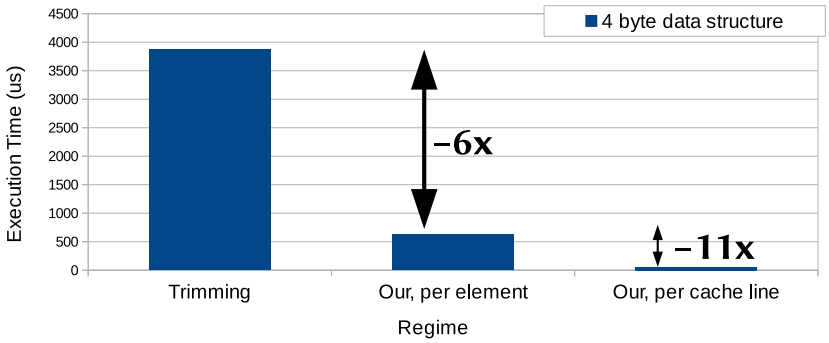



Figure 8: The execution time of the prefetch phase under three different *Prefetch phase* regimes.

validate the correctness of all blocks of the proposed toolchain and to evaluate its performance, we created several batches of experiments based on the composition of five ADAS inspired kernels. Such batches are instantiated with different parameters to create a number of use-case scenarios. For given scenarios, we generated PREM compliant code by using the proposed compiler, profiled the resulting code to get execution times of generated PREM intervals, generated a schedule by using the ILP solver or the heuristic, and run experiments on NVIDIA Jetson TX1 board (based on ARM Cortex A57 processor). We use Linux 3.16 to run the experiments, and to establish the predictable behavior required for the PREM model, we implemented system calls for temporary disabling / enabling of interrupts on the selected core and for flushing and invalidating the entire cache. We measured execution times and the numbers of cache misses in particular intervals by using the performance monitor unit (event `L2D_CACHE_REFILL` and `PMCCNTR` register) in user space. The ILP model solved in IBM ILOG CPLEX Optimization Studio or the heuristic algorithm implemented in C# gives start times which define sequencing of memory intervals in our test bed.

### *6.1. Kernels*

For the evaluation, five kernels are used, from which different scenarios based on real use cases are constructed. The first three kernels were already described in Section 2 (GEMM, FFT, and binary tree search). In addition to this 2D convolution (2DConv) and 2D Jacobi stencil computation (2DJacobi) were adopted from PolyBench/ACC benchmark suite [14]. The 2D convolution is widely used in signal processing and machine learning, and the 2D Jacobi stencil can be used for instance to solve a system of linear equations. 2D Jacobi consists of two kernels, and it is important to note that the second kernel (2DJacobi-2) is strictly a data copy kernel, i.e., it has no computation. Overall, these kernels have different memory access patterns, compute-to-communication ratios (CCR), which allows us to draw further conclusions on the effects of PREM.

### *6.2. Compiler Optimizations*

The creation of prefetch phases has been proposed previously in [15, 16], to separate the memory operations from computations for different reasons. In these previous works, the prefetch phases have been created through the reuse of the original control flow from the computatation part, a process that can be thought of as *trimming* the original code for a specialized purpose. This *trimming* approach has several drawbacks, as the control flow may be overly complex for just performing prefetching, and if the compiler is unable to identify the complexity and optimize it out, it may lead to long execution times for the prefetch operations. Furthermore, repeating accesses over multiple iterations of a loop may lead to the same data being prefetched multiple times, in for example stencil-type kernels, in which each loop iterations accesses its neighbor elements. In addition to this, this approach turns every load in the original code into a prefetch operation, disregarding the optimization of just performing one prefetch per cache line, as presented in Section 3.3.

To illustrate the effects of this, we executed the previously presented 2DConv kernel, which has stencil-type accesses, and measured the execution time of the prefetch phases under three different regimes: *trimming*, our approach without cache line optimizations, and our approach with cache line optimizations. The execution was performed on the NVIDIA TX1, and the results are presented in Figure 8. As can be seen, for this kernel, the transformation presented in this approach is 6× faster than *trimming* when not considering that multiple elements may be part of the same *cache line*. On top of this, the *cache line* optimization further increases the performance of the *Prefetch phase* 11×. We will explain this effect in further detail, beginning with the 6× improvement compared to trimming. The kernel considered is a $3 \times 3$ convolution kernel, meaning that during the execution of the kernel, each element will be accessed 9 times (disregarding elements on the border). Once as the center element, and eight times for the neighbor direction (north-west, north, north-east, east, ...). Therefore, reusing the original control flow to prefetch this data means that each element is fetched 8 times more than needed. With our approach, we identify the exact memory access pattern, and can produce a *Prefetch phase* that only visits each element once. Thus, the upper bound on the improvement from this transformation is 8×, due to the removal of duplicate accesses, and the measured improvement is 6×.

To understand the 11× improvement when optimizing for cache line reuse, we start by realizing that the data types accessed in the kernel are *floats* of 4 bytes, and that the NVIDIA TX1 has a cache line size of 64 bytes. This means that each cache line contains 16 floats stored sequentially in memory. Since data is moved to the cache at a *cache line* granularity, prefetching any element of a cache line will automatically load the remaining elements. Thus, for sequential accesses our approach increases the stride of the prefetch loop to only touch one element per cache line, leading to an upper bound for this optimization of 16×, of which we measure an improvement of 11×. Note that the size of the elemements accessed, and the size of a *cache line* of the system directly influences the expected gains of this optimization, although these values are fairly typical. Combining these two optimizations, our approach is able to deliver almost 70× the performance of the *trimming* approach used in previous works, for kernels that have a high degree of duplicate accesses.

### *6.3. Kernel Characterization*

As the current COTS processors are optimized for average-case performance, it is difficult to obtain realistic estimation of worst-case execution time by sequential execution of a scenario. To underline the importance of scheduling of memory accesses and to get closer to realistic WCETs of our scenarios, we evaluated sensitivity of our kernels to a memory interference generated by memory intensive tasks on other cores. The knowledge of sensitivity of the kernels also shows theoretical profitability of the conversion of a scenario into a PREM compatible form.

To provide insights on the effects of PREM on the execution times, we consider two cases: The average case execution time (ACET) and, more importantly, the WCET. While the ACET is not important for real-time scheduling, nor provides a realistic expectation on achievable performance under real-time guarantees, it provides an insight on the performance effects of the PREM transformations done within the compiler.

Figure 9 shows the execution times of the PREM transformed code, normalized to the code without PREM transformations, which we refer to as Legacy code. The first thing to notice is that, when not considering the Writeback phase required due to the random cache replacement policy, PREM performance is always better than Legacy. This happens, because the tiling transformation done by the compiler improves the data locality of the transformed code, in the same manner as for the well known blocked matrix multiplication [17]. Furthermore, the prefetch phase improves the memory bandwidth utilization.

When including the Writeback phase, PREM shows different amount of overhead for the kernels. Since all transformed kernels load the same amount of data (half of the LLC capacity), the amount of compute operations that can be performed on that data before a refill is needed dictates how much of the execution time is spent on data management (writeback). Because of the random cache replacement policy, the cost of the writeback phase is quite expensive, as each cache line must be individually flushed, which is a high-latency operation. For Legacy code, these operations are not necessary, meaning that all time spent on cache flushes implies overhead for PREM. In kernels with a high CCR (such as GEMM mul), the larger compute phase dominates this cost, leading to a relatively small overhead. As can be seen in the figure, the Writeback phase is only a small fraction of the overall execution time. However, in kernels with a low CCR

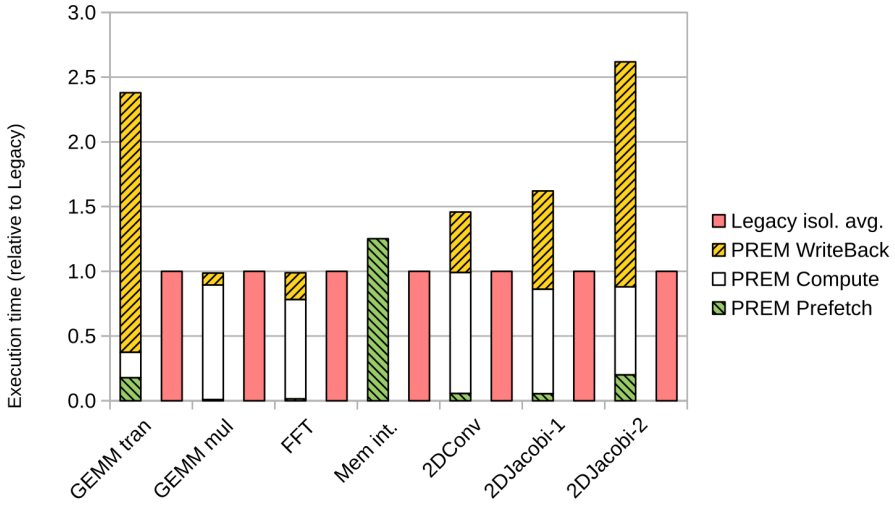



Figure 9: Normalized execution times of kernels

(such as 2DJacobi-2), the Writeback phase makes up around two thirds of the execution time. In other words, the kernel is almost 3× slower due to cache management. Thus, for best PREM performance, the kernels should have a high CCR, enabling improved performance through guaranteed data locality.

With the presented ACET for the kernels, it is clear that the performance degradation due to memory interference needs to be higher for kernels whose performance degrades under PREM, for PREM to provide an improved WCET. For this reason, we next measure the WCET for Legacy and PREM. To quantify the WCET of the kernels under memory interference, we exposed our kernels to interfering tasks with two types of memory access patterns. The first is random memory access pattern, the second is sequential memory read. We compare execution times of PREM intervals with WCETs of equivalent legacy codes. In practice, we run compute phases of PREM intervals without prefetch and write-back phases and with interfering tasks on other cores. In PREM, there is no space for interfering tasks. During memory phases, the core has exclusive access to the main memory, and execution time of a compute phase is not affected by competing for main memory, because all required data are already in core-local memory.

We compare the ratio between WCET of legacy intervals and PREM intervals in Figure 10, with memory intensive tasks on other cores. We consider only the worst behaving interference pattern. All values are normalized to the Legacy execution time without interference (same as Figure 9). There are three main observations in this experiment. First, the overheads caused by writebacks are significant, and even under memory interference not all kernels benefit from conversion into predictable intervals. This is once again particularly clear for the 2DJacobi-2 kernel. Second, interfering tasks can cause significant slowdown (up to 2.6× for mmul). In cases where the slowdown due to interference is significant, even cases where the writeback phase causes performance degradation in PREM, the WCET is improved. Third, while the Legacy code is affected by memory interference, the PREM code remains the same, which is due to the isolation property of PREM memory scheduling. Sequential memory accesses were more interfering for all intervals except the memory intensive interval, where random memory accesses caused significantly bigger slowdown. Overall, these results are in line with previous results observed in [18].

In this paper, we use only the CPU cluster of the TX1 platform for our experiments. In the future work we consider an extension of the PREM to both CPU and GPU clusters, and as was shown in [16], slowdowns on the GPU side can be even much worse than slowdowns on CPUs.

### *6.4. Use-case scenarios*

We evaluate the effectiveness of our toolchain on several application execution scenarios, composed of different combinations of the previously presented kernels. Four small scenarios (each scenario having

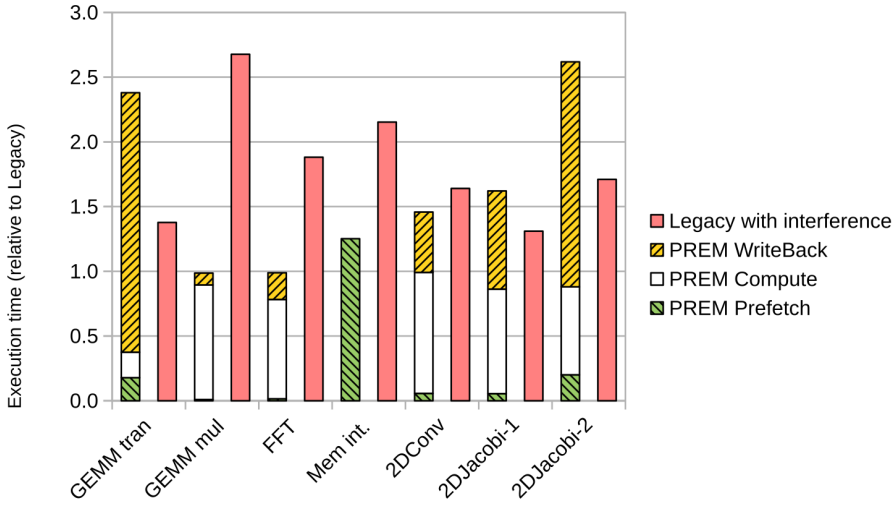



Figure 10: Kernel execution times under interference

only sixteen intervals), and two large scenarios (each having up to five hundred intervals) are used. We differentiate between small and large by the feasibility to solve the scheduling problem with the ILP approach.

The test scenarios are constructed and compiled to generate the following control flows: For each GEMM, one chain of transpositions and subsequent multiplications that can run in parallel are generated. For the 2D convolution, several parallel intervals are generated and similarly for 2D Jacobi, where two subsequent parallel intervals (2DJacobi-1 and 2DJacobi-2) are generated. For these intervals, the quantity of generated intervals depends of the data size. The FFT and tree traversing are fixed-size and stand-alone intervals in our scenarios. All instances are solved by the heuristic algorithm, but only small instances are solved by the ILP solver, as it is not able to solve large instances in reasonable time.

Table 1 describes compositions of small scenarios. Each application of a scenario is described by two numbers – count and parallelism. We explain the meaning of these numbers on *Scn. 1*, which is the scenario from Figure 1 where three applications are run in parallel. The first application consists of two subsequent GEMMs ($C = \alpha A \times B + \beta C$), hence we have GEMM *count* of 2. The intervals $I_1$ and $I_6$ are transpositions of the matrix $B$ and $I_{2,3,4,5}$ and $I_{7,8}$ are actual multiplications that can run in parallel, hence GEMM *parallelism* is 4 and 2 respectively. The second application is FFT followed by inverse FFT (FFT count 2, parallelism 1), and the third application is binary search tree algorithm divided into multiple intervals (Search count 5, parallelism 1). The four selected small scenarios are:

1. the scenario in Figure 1, explained above,
2. the second scenario is composed of exactly the same applications, the only difference is a division of binary searches into two parallel chains of intervals ($I_{12}$, $I_{13}$, $I_{14}$ and $I_{15}$, $I_{16}$),
3. the third scenario has only one multiplication divided into seven parallel intervals and
4. the fourth scenario has the same two GEMMs as in Scn. 1, two independent FFTs followed by inverse FFTs and only three graph traversal intervals.

The number of parallel multiplications was automatically generated by the compiler which converted all scenarios into PREM-compliant code. The amount of data processed by FFT was selected such that the data completely fits into core-local memory. Binary tree search intervals cannot be efficiently converted into predictable intervals, therefore we marked them for transformation into compatible intervals. The compiler also generated scenarios with uncontrolled access to main memory by taking the same dependency graphs and intervals without prefetch and write-back phases. As before, we call these codes *Legacy*.

Two large scenarios (Scn. 5 and Scn. 8×1) are used to evaluate the heuristic. Scenario 8×1 is created by duplicating Scenario 1 eight times in sequence. This makes the scenario too large to solve with the ILP solver, but since we can use the ILP solver to find an exact solution for each part, we know the optimal

| Scenario | Scn. 1 | Scn. 2 | Scn. 3 | Scn. 4 | Scn. 5 | Scn. 8×1 |
|---|---|---|---|---|---|---|
| GEMM count | 2 | 2 | 1 | 2 | 10 | 16 |
| GEMM parallelism | 4, 2 | 4, 2 | 7 | 4, 2 | see Fig. 11 | 4, 2, ... |
| FFT count | 2 | 2 | 2 | 4 | 4 | 16 |
| FFT parallelism | 1 | 1 | 1 | 2 | 2 | 1 |
| Search count | 5 | 5 | 5 | 3 | 40 | 40 |
| Search parallelism | 1 | 2 | 1 | 1 | 7 | 1 |

Table 1: Composition of evaluated scenarios

completion time of the sequential composition into a large scenario. This allows us to evaluate the heuristic scheduler performance on large scenarios against a known optimal solution. The second large scenario (Scn. 5) is inspired by real-world applications, that might be executed together in practice. In detail, we take inspiration from a KCF tracker [19] (*tracker*), convolutional neural networks (*neural*), control tasks (*control*), and image processing pipelines (*image*). An overview of the components of this scenario is shown in Figure 11. The parallelism of the kernels is expressed as a number after the name (e.g. GEMM 4 contains one transposition and four parallel multiplications kernels). On top of the figure are two parallel chains inspired by the *tracker*. These chains start and end with a memory intensive task, e.g. opening an image file. Each parallel chain consists of a convolution, a memory intensive interval, an FFT, a matrix multiplication, a iFFT and another memory intensive interval. Below the *tracker* chains is an interval chain inspired by the *neural* application, which is executed in parallel with the *tracker*. This chain consists of matrix multiplications used in evaluation of a neural network. The next chain, below the neural network, represents the *control* application, and consists of Jacobi kernels for solving a system of linear equations. At the bottom of the figure are 4 chains of memory intensive tasks, such as graph traversing or binary tree searches, which represent the *image* application. Overall, this task combines the components of a possible ADAS-style system, where *image* represents the acquisition of image data, *tracker* and *neural* represent the processing of this data, and lastly *control* represents the actuation on the system.

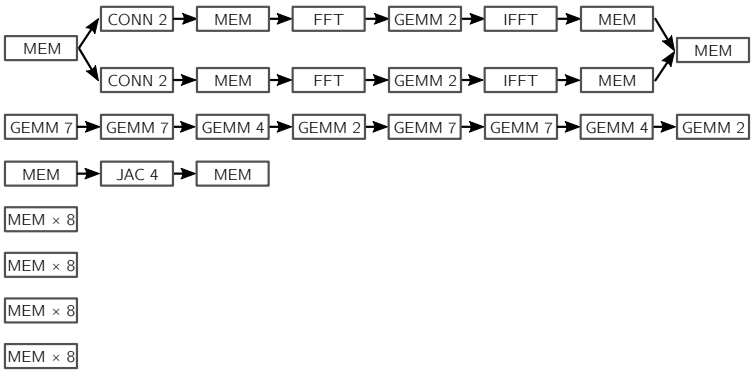



Figure 11: Scenario 5 consisting of 111 intervals. Numbers represent the parallelism of the kernel.

Execution times of particular PREM phases were obtained by taking the worst-case execution time from 100 executions on a single core. Then we solved the ILP model (for small scenarios) and executed the heuristic with the obtained execution times.

We evaluate our PREM compliant scenarios executed according to the solved schedules on 100 000 runs and compare that with an implementation with uncontrolled access to the main memory. Both implementations are based on a thread pool in order to minimize overheads for creating new threads. Jobs to be executed by the threads are picked from a queue. In PREM execution, the pool has a thread for each CPU core and the queue is ordered according to the schedule. When a PREM phase finishes earlier than expected, the subsequent phase is executed immediately once all dependencies are satisfied. In *Legacy* executions, the queue is dynamically filled based on the DAG and the jobs are executed by idle threads. The total number of threads equals to the maximum parallelism achievable in the application. All threads are scheduled by

| | | Small | | | | Large | |
|---|---|---|---|---|---|---|---|
| Scenario | | Scn. 1 | Scn. 2 | Scn. 3 | Scn. 4 | Scn. 5 | Scn. 8×1 |
| Number of intervals | | 16 | 16 | 16 | 16 | 111 | 128 |
| ILP PREM | $C_{\mathrm{MAX}}$ (ms) | 7.92 | 7.92 | 8.42 | 8.10 | – | – |
| | WCET (ms) | 7.79 | 7.79 | 8.40 | 8.09 | – | – |
| | Mean (ms) | 7.63 | 7.63 | 8.32 | 7.87 | – | – |
| Heuristic PREM | $C_{\mathrm{MAX}}$ (ms) | 8.07 | 8.40 | 9.08 | 8.43 | 79.20 | 73.40 |
| | WCET (ms) | 8.03 | 8.15 | 8.56 | 8.32 | 79.10 | 73.24 |
| | Mean (ms) | 7.77 | 8.00 | 8.36 | 8.07 | 78.19 | 70.27 |
| Legacy | WCET (ms) | 9.75 | 11.27 | 11.09 | 10.79 | 96.30 | 88.27 |
| | Mean (ms) | 7.77 | 9.11 | 8.92 | 8.93 | 72.31 | 60.46 |
| ILP solving time (s) | | 41 | 73 | 7 908 | 60 | – | – |
| Heuristic solving time (s) | | 0.9 | 1.6 | 0.8 | 0.6 | 53 | 46 |

Table 2: Scheduled completion time and measured execution times for the scenarios, as well as the time required to find schedules.

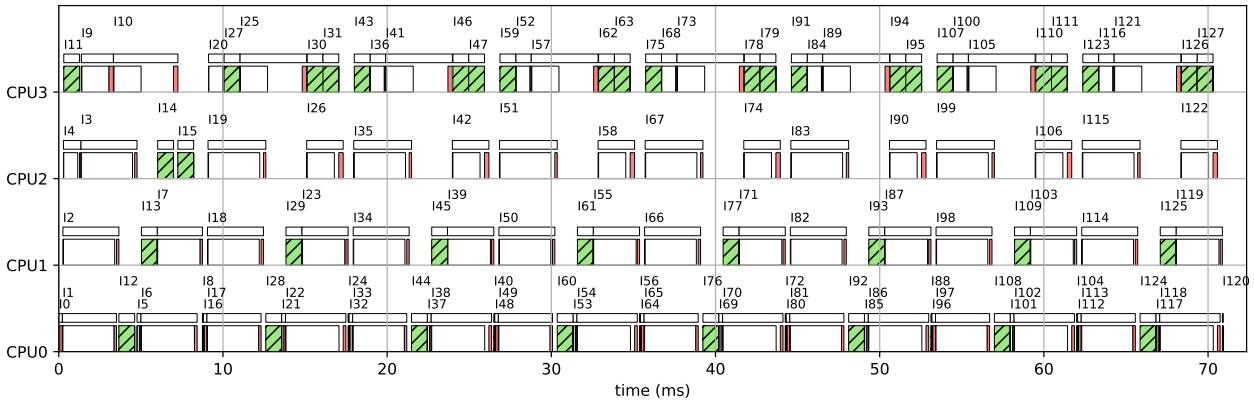



Figure 12: Gantt diagram of the 8×1 scenario schedule. Intervals in the schedule are represented by white rectangles with labels $I_i$ (e.g. $I_{11}$ or $I_9$ in top left corner). Below the intervals are rectangles representing phases of the intervals. Predictable intervals have three phases – two red for memory and a white compute phase, compatible intervals have only one phase – hatched green)

the Linux `SCHED_FIFO` scheduler and have the same priority.

### *6.5. Experimental results*

In Table 2 the measured worst-case execution times (WCET) and mean execution times are shown for each of the four small scenarios, both for PREM and *Legacy* executions. Furthermore, the schedule completion times $C_{\mathrm{MAX}}$ calculated by the ILP solver and the heuristic are shown for the PREM schedules (*Legacy* schedules are based on best-effort and have no pre-determined schedules). Lastly, the time required to find the optimal and the heuristic schedule for each of the scenarios is provided.

The ILP solver was able to find a solution for up to 34 activities (16 PREM intervals) in a reasonable time (last line in Table 2 shows solution times on Intel Core i7-3770). In three of the four cases, the ILP solver was able to find a solution in less than two minutes. In the last case, the exploration took longer, due to the significantly larger solution space caused by additional parallelism in the task set. The solution of Scenario 1 in the form of Gantt diagram is in Figure 12.

The measured execution times of all 100 000 runs of our small scenarios are presented in logarithmic scale histograms in Figures 13a–13d. The PREM schedules completion times $C_{\mathrm{MAX}}$ are shown as a dashed vertical lines.

There are three main findings in the results of the experiments. First, in every scenario, the variance of completion times under PREM is small (max 6.1%) in comparison to *Legacy* executions (up to 52.4%). We calculate the variance $P$ for PREM as $P = 100 \times (WCET_{\mathrm{PREM}}/BCET_{\mathrm{PREM}} - 1)$ where $WCET_{\mathrm{PREM}}$

and $BCET_{\text{PREM}}$ are the measured worst and best case execution times of the PREM compliant execution and analogously $L = 100 \times (WCET_{\text{Legacy}}/BCET_{\text{Legacy}} - 1)$ for the *Legacy* execution. Higher variances of *Legacy* executions are caused by non-optimal schedules resulting from dynamic scheduling algorithm as well as by competition for the shared memory. For example, we can see in Figure 13d that the histogram of the *Legacy* executions has three major peaks which correspond to three different schedules and selection of particular schedule depends on actual execution times of preceding intervals. If an interval is delayed, then a different schedule is selected at runtime. We can clearly see the positive impact of PREM in combination with static scheduling on the variance of completion times. The variance could be even smaller if we strictly followed start times of the generated schedule.

Second, the measured WCET of the PREM schedule is always smaller than calculated schedule completion time. Since we allow execution of intervals as soon as they are ready (we do not wait for the corresponding start time when all dependencies are satisfied and requested resources are available), the whole scenario can finish earlier. The fact that all executions finish before the estimated WCET shows that our WCET estimations of particular tasks acquired by single core profiling are sufficient and are not affected by the execution of multiple intervals on a multi-core system at the same time.

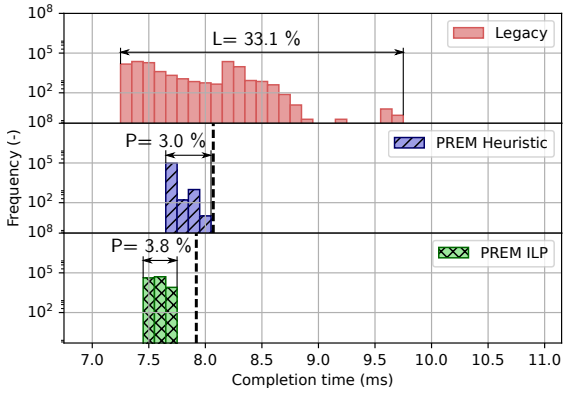
Third and most important, the measured WCET of PREM executions is always smaller than the WCET of *Legacy* executions (at least by 25.1%, and up to 44.7% for exact solutions, and at least by 21.4%, and up to 38.3% for solutions produced by the heuristic). We calculate the WCET difference as $WCET_P^L = 100 \times (WCET_{\text{Legacy}}/WCET_{\text{PREM}} - 1)$. The WCET of *Legacy* executions is strongly affected by dynamic scheduling algorithm which does not understand the structure of the scenario. For example scenarios 1 and 2 are composed of the same intervals, the only difference is that Scenario 2 enables execution of two memory intensive intervals at the same time. Concurrent execution of the intervals ($I_{12}$ and $I_{15}$) prolongs both of them up to three times as can also be seen in Table 3, and therefore subsequent tasks are significantly delayed. As can be seen in Figure 13b, the delay influences the WCET of the *Legacy* execution which is 11.27 ms instead of 9.75 ms as well as the mean time which is 9.11 ms instead of 7.77 ms while the optimal static schedule for PREM model is the same in both scenarios.

We can also observe that on our small instances, the heuristic performs well, and produces schedules only about 5 to 10 % slower than the optimal solutions, while, according to Table 2, computation of the schedule with the heuristic is 45–8000× faster than the ILP approach. Also, the average execution times as well as the worst-case execution times are always better than in *Legacy* executions.

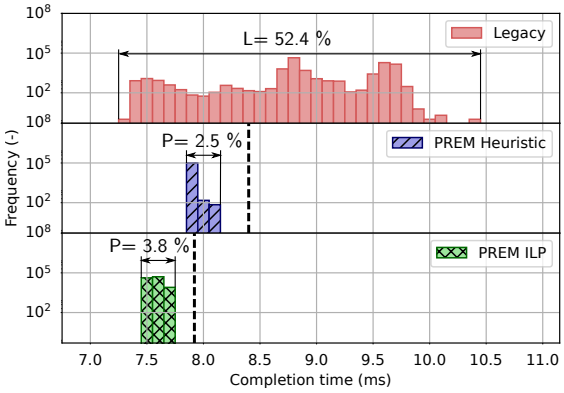
For larger scenarios 5 and 8×1, where the optimal solver was not applicable, the heuristic is the only option to produce the schedule. The benefit of PREM on large scenarios depends on many factors. The sensitivity to memory interference plays an important role, which was already discussed in Section 6.3, as does the structure of the scenario and the efficiency of the heuristic. We show the execution time histograms of the large scale scenarios in Figure 14. It can not be easily determined how far is the generated solution from the optimal solution. For this reason, we use Scenario 8×1, with the histogram depicted in Figure 14b, consisting of eight times sequentially duplicated Scenario 1, for which we already know the optimal solution. Therefore, for this particular scenario, we can compare generated schedule with the optimal one. Optimal $C_{MAX}$ for the scenario equals $8 \times 7.9 = 63.2$ while the heuristic-generated schedule has $C_{MAX} = 73.4$. This represents 15.5 % increase.

The histogram of execution times of Scenario 5 is shown in Figure 14a. While the average execution time of PREM (see Tab. 2) is higher than that of *Legacy*, the measured WCET of *Legacy* is much larger than in the PREM execution ($WCET_P^L = 21.75\,\%$). The reason for the higher average execution time is that this scenario includes several kernels that were already shown have smaller performance under PREM (see Section 6.3), however, even in light of this, PREM provides tighter WCET bounds in this scenario. This is a good outcome, as the WCET is the limiting factor in how many tasks that can be successfully scheduled in a system. Also, we can see that the execution time variance is much lower in the PREM execution (1.5 % vs 65.1 %). From this we see that PREM successfully reduces the execution time jitter, greatly improving the predictability of the system.

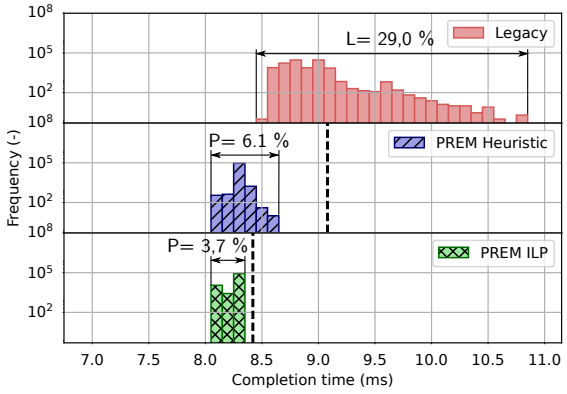
Table 3 shows the measured execution times and number of cache misses in Scenarios 1 and 2. Each predictable interval has measurements shown for each of the PREM phases (Prefetch, Compute and Write-back). For compatible intervals, the measured values are in the prefetch column only, as compatible intervals



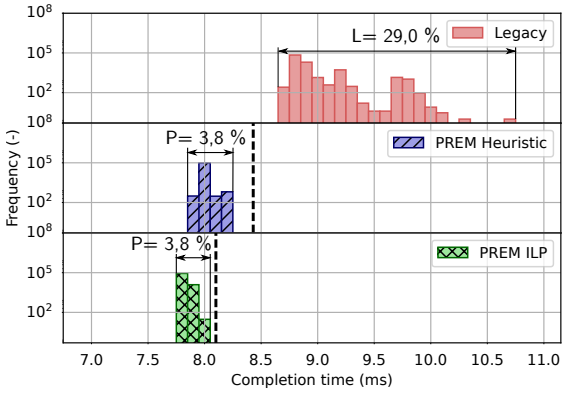

(a) Scenario 1




(b) Scenario 2

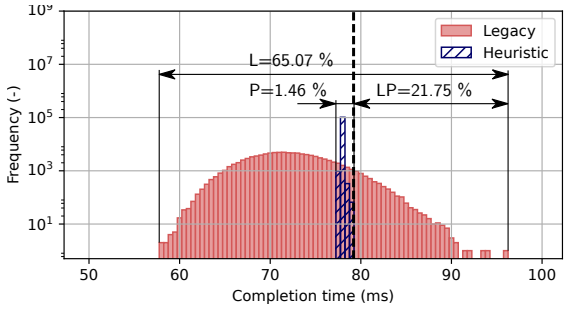



(c) Scenario 3

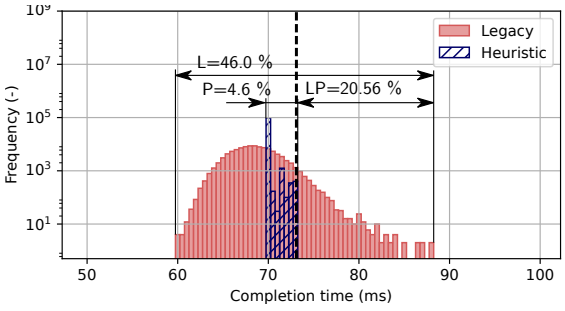



(d) Scenario 4

Figure 13: Histograms comparing completion times of small scenarios with and without PREM applied




(a) Scenario 5




(b) Scenario 8×1

Figure 14: Histograms comparing completion times of large-scale scenarios with and without PREM applied

| | PREM | | | | | | Legacy Scn. 1 | | Legacy Scn. 2 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Time (us) | | | Cache misses | | | Time | Cache | Time | Cache |
| | P | C | W | P | C | W | (us) | miss. | (us) | miss. |
| $I_1$ | 28 | 31 | 162 | 3 454 | 22 | 0 | 106 | 3 510 | 82 | 3 519 |
| $I_2$ | 35 | 3 106 | 145 | 4 063 | 12 | 0 | 3 188 | 4 914 | 3 180 | 4 982 |
| $I_3$ | 34 | 3 108 | 145 | 4 063 | 13 | 0 | 3 188 | 4 970 | 3 187 | 5 055 |
| $I_4$ | 33 | 3 188 | 146 | 4 071 | 15 | 0 | 3 651 | 5 014 | 3 211 | 5 380 |
| $I_5$ | 20 | 847 | 78 | 2 380 | 19 | 4 | 866 | 2 504 | 1 127 | 2 547 |
| $I_6$ | 16 | 23 | 93 | 1 901 | 9 | 0 | 91 | 1 930 | 43 | 1 971 |
| $I_7$ | 32 | 3 198 | 166 | 4 079 | 9 | 0 | 3 595 | 4 088 | 3 245 | 4 585 |
| $I_8$ | 28 | 2 548 | 138 | 3 459 | 15 | 0 | 2 652 | 3 930 | 2 603 | 3 540 |
| $I_9$ | 55 | – | – | 255 | – | – | 45 | 250 | 46 | 263 |
| $I_{10}$ | 35 | 1 667 | 277 | 4 096 | 22 | 0 | 2 324 | 5 790 | 2 500 | 5 811 |
| $I_{11}$ | 34 | 1 670 | 275 | 4 081 | 21 | 0 | 2 308 | 6 281 | 2 361 | 6 428 |
| $I_{12}$ | 877 | – | – | 3 850 | – | – | 862 | 4 942 | 2 355 | 4 529 |
| $I_{13}$ | 860 | – | – | 3 800 | – | – | 788 | 4 456 | 1 555 | 4 058 |
| $I_{14}$ | 862 | – | – | 3 805 | – | – | 794 | 4 432 | 784 | 4 145 |
| $I_{15}$ | 867 | – | – | 3 802 | – | – | 756 | 4 009 | 2 343 | 4 434 |
| $I_{16}$ | 858 | – | – | 3 800 | – | – | 754 | 3 911 | 1 527 | 4 324 |

Table 3: Sample of measured execution times and cache misses for scenarios 1 and 2

only consist of a single memory phase.

From the table two important results can be seen for the memory isolation property of PREM. First, the compute phases of the PREM-compliant executions have a negligible amount of cache misses, even though the compiler does not prefetch stack data, and the cache employs a random replacement policy. This means that even under these conditions, the proposed toolchain is able to produce both a system schedule and transform the code such that the memory isolation property of PREM is upheld in practice.

Second, it can be seen that the memory phases of the PREM-compliant executions show an average of 15% fewer cache misses. We believe this is due to the explicit eviction of data that is no longer used, such that the loading of new data is less likely to evict newly loaded data due to the random replacement policy.

## 7. Related work

The predictable execution model was originally proposed and evaluated on a single core processor by R. Pellizzoni et al. [1]. The first attempt to extend PREM to multi-core systems was made by S. Bag et al. [20]. Although in these papers a conceptual definition of a compiler for automatic generation of PREM-compliant code is provided, no real implementation is discussed. Concerning task scheduling, the authors simulated behavior of traditional dynamic schedulers, such as rate monotonic or earliest deadline first, applied to synthetically generated PREM scenarios. Subsequently G. Yao et al. [21, 22] proposed memory-centric scheduling technique that employs time division multiple access to shared memory and enables preemption of PREM predictable intervals. A. Alhammad and R. Pellizzoni [23] proposed static scheduling heuristic for PREM compliant fork-join tasks. All the above papers assume caches with deterministic replacement policies as local memories, and evaluations are based on simulations or on execution on x86 platforms. Overall, our paper is the first to describe fully-integrated PREM-support for state-of-the-art multi-core embedded CPUs, with a realistic setup running on real hardware and considering real-life benchmarks. Several other papers such as A. Alhammad et al. [24] or Burgio et al. [25] utilize scratch-pad memories (SPM). Unfortunately, many multi-core embedded platforms (such as NVIDIA TX1 used in our paper) have only cache memories with non-deterministic replacement policies and do not have explicitly managed memories.

Manual conversion of an application into PREM compliant format is time-consuming. Therefore the original PREM paper [1] converts manually marked functions automatically at the compiler level. A compiler independent solution based on memory profiling tools and backward refactoring of manually selected parts of the code was proposed R. Mancuso et al. [13]. However, no fully automated tool for transformation of code into PREM compliant code exists so far. Our compiler, although still not fully mature, is capable of handling legacy codes written in compliance to standard automotive coding best practices. A related problem was addressed by Koukos et al. [15] who employ an execution model similar to PREM to minimize power consumption. The main idea is separation of memory phase and lowering CPU frequency during

the prefetch phase. While this work shared the underlying concept of memory/compute separation, the application is completely different, as are the practical challenges.

PREM is not the only mechanism able of achieving predictable execution on COTs components based systems. MemGuard proposed by H. Yun et al. [26] is a memory bandwidth reservation system that provides guaranteed bandwidth for temporal core isolation. Another way to achieve predictability can be DRAM bank-aware allocation proposed by H. Yun [27]. However, on some platforms (such as NVIDIA TX1), controlling DRAM bank allocation is problematic due to address randomization aimed at improving average performance. These approaches can be considered as orthogonal to what we describe here. The integration of PREM compilation and bandwidth reservation on top of static schedules can provide additional benefits.

The use of integer linear programming has long tradition in the development of parallel automotive real-time systems. For example, Becker et al. [28] propose a contention-free execution framework evaluated on an AUTOSAR-based engine management unit. They use both ILP and heuristic algorithms to find static schedules. Their approach to application scheduling is similar to ours, with the main difference being that we have actually evaluated the results by executing the application on real hardware.

## 8. Conclusion

In this paper, we proposed a toolchain for automated code transformation of parallel applications into PREM-compliant structure and their execution on multi-core homogeneous system according to the static schedule obtained by either solving an integer linear programming model or an efficient heuristic. Experimental evaluation shows that for the selected ADAS-like scenarios, PREM in combination with static scheduling brings the following benefits: i) Significant (at least 5 times) reduction of completion time jitter, ii) WCET of the PREM schedule is always smaller than the calculated schedule completion time and iii) the measured WCET of PREM executions is smaller than the WCET of legacy executions on most scenarios. The adopted heuristic performs well on small instances (5–10% above the optimum), and it is capable of solving large-scale problems in reasonable time of few seconds.

Our ultimate goal is to improve predictability of execution on systems with integrated parallel accelerators such as GPU-based systems-on-a-chip (SoC) or Xeon Phi. PREM has been demonstrated to improve predictability of execution on multi-core CPUs (this work) or integrated GPUs [16]. Our current and future work focuses on putting all these pieces together, and enable off-line scheduling-based whole-system control of predictable execution for this type of SoCs, which are more and more used as a target for time-critical applications (e.g., ADAS, avionics). In addition to this, we are also conducting an exploration of the PREM execution on an Intel machine with Cache Allocation Technology, where we can explicitly partition the last level cache and assign cache partitions to individual cores. This enables finer control of the core-local memory and provides better isolation properties between the cores, and as such it presents an attractive target for time-critical applications.

## 9. Acknowledgement

This work was supported by the HERCULES Project, funded by European Unions Horizon 2020 research and innovation program under grant agreement No. 688860.

## References

## References

[1] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, R. Kegley, A predictable execution model for cots-based embedded systems, in: 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, 2011, pp. 269–279 (April 2011). doi:10.1109/RTAS.2011.33.

[2] A. Bastoni, B. B. Brandenburg, J. H. Anderson, Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability, in: Proc. $6^{th}$ International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010), Brussels, Belgium, 2010 (Jul. 2010).

[3] C. Lattner, V. Adve, Llvm: a compilation framework for lifelong program analysis transformation, in: International Symposium on Code Generation and Optimization, 2004. CGO 2004., 2004, pp. 75–86 (March 2004). doi:10.1109/CGO.2004.1281665.

[4] J. Matějka, B. Forsberg, M. Sojka, Z. Hanzálek, L. Benini, A. Marongiu, Combining prem compilation and ilp scheduling for high-performance and predictable mpsoc execution, in: Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'18, ACM, New York, NY, USA, 2018, pp. 11–20 (2018). doi:10.1145/3178442.3178444.
URL http://doi.acm.org/10.1145/3178442.3178444

[5] Z. Hanzálek, P. Šůcha, Time symmetry of resource constrained project scheduling with general temporal constraints and take-give resources, Annals of Operations Research 248 (1) (2017) 209–237 (Jan 2017). doi:10.1007/s10479-016-2184-6.
URL https://doi.org/10.1007/s10479-016-2184-6

[6] Z. Hanzálek, A parallel algorithm for gradient training of feedforward neural networks, Parallel Comput. 24 (5-6) (1998) 823–839 (Jun. 1998). doi:10.1016/S0167-8191(98)00035-0.
URL http://dx.doi.org/10.1016/S0167-8191(98)00035-0

[7] J. Liedtke, H. Hartig, M. Hohmuth, Os-controlled cache predictability for real-time systems, in: Proceedings Third IEEE Real-Time Technology and Applications Symposium, 1997, pp. 213–224 (Jun 1997). doi:10.1109/RTTAS.1997.601360.

[8] B. Forsberg, A. Marongiu, L. Benini, Gpuguard: Towards supporting a predictable execution model for heterogeneous soc, in: DATE'17, 2017 (2017).

[9] M. R. Soliman, R. Pellizzoni, WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching, in: M. Bertogna (Ed.), 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), Vol. 76 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017, pp. 24:1–24:23 (2017). doi:10.4230/LIPIcs.ECRTS.2017.24.
URL http://drops.dagstuhl.de/opus/volltexte/2017/7175

[10] M. I. S. R. Association, M. I. S. R. A. Staff, MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems, Motor Industry Research Association, 2013 (2013).
URL https://books.google.ch/books?id=3yZKmwEACAAJ

[11] T. Grosser, A. Groesslinger, C. Lengauer, Polly – performing polyhedral optimizations on a low-level intermediate representation, Parallel Processing Letters 22 (04) (2012). arXiv:http://www.worldscientific.com/doi/pdf/10.1142/S0129626412500107, doi:10.1142/S0129626412500107.

[12] 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, P. Stenström, The worst-case execution-time problem—overview of methods and survey of tools, ACM Trans. Embed. Comput. Syst. 7 (3) (2008) 36:1–36:53 (May 2008). doi:10.1145/1347375.1347389.
URL http://doi.acm.org/10.1145/1347375.1347389

[13] R. Mancuso, R. Dudko, M. Caccamo, Light-prem: Automated software refactoring for predictable execution on cots embedded systems, in: 2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications, 2014, pp. 1–10 (August 2014). doi:10.1109/RTCSA.2014.6910515.

[14] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, J. Cavazos, Auto-tuning a high-level language targeted to gpu codes, in: 2012 Innovative Parallel Computing (InPar), 2012, pp. 1–10 (May 2012). doi:10.1109/InPar.2012.6339595.

[15] K. Koukos, P. Ekemark, G. Zacharopoulos, V. Spiliopoulos, S. Kaxiras, A. Jimborean, Multiversioned decoupled access-execute: The key to energy-efficient compilation of general-purpose programs, in: Proceedings of the 25th International Conference on Compiler Construction, CC 2016, ACM, New York, NY, USA, 2016, pp. 121–131 (2016). doi:10.1145/2892208.2892209.
URL http://doi.acm.org/10.1145/2892208.2892209

[16] B. Forsberg, L. Benini, A. Marongiu, Heprem: Enabling predictable gpu execution on heterogeneous soc, in: 2018 Design, Automation Test in Europe Conference Exhibition (DATE), 2018, pp. 539–544 (March 2018). doi:10.23919/DATE.2018.8342066.

[17] M. D. Lam, E. E. Rothberg, M. E. Wolf, The cache performance and optimizations of blocked algorithms, in: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IV, ACM, New York, NY, USA, 1991, pp. 63–74 (1991). doi:10.1145/106972.106981.
URL http://doi.acm.org/10.1145/106972.106981

[18] R. Cavicchioli, N. Capodieci, M. Bertogna, Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms, in: 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2017, pp. 1–10 (Sept 2017). doi:10.1109/ETFA.2017.8247615.

[19] J. F. Henriques, R. Caseiro, P. Martins, J. Batista, High-speed tracking with kernelized correlation filters, CoRR abs/1404.7584 (2014). arXiv:1404.7584.
URL http://arxiv.org/abs/1404.7584

[20] S. Bak, G. Yao, R. Pellizzoni, M. Caccamo, Memory-aware scheduling of multicore task sets for real-time systems, in: 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2012, pp. 300–309 (Aug 2012). doi:10.1109/RTCSA.2012.48.

[21] G. Yao, R. Pellizzoni, S. Bak, E. Betti, M. Caccamo, Memory-centric scheduling for multicore hard real-time systems, Real-Time Systems 48 (6) (2012) 681–715 (Nov 2012). doi:10.1007/s11241-012-9158-9.
URL https://doi.org/10.1007/s11241-012-9158-9

[22] G. Yao, R. Pellizzoni, S. Bak, H. Yun, M. Caccamo, Global real-time memory-centric scheduling for multicore systems, IEEE Transactions on Computers 65 (9) (2016) 2739–2751 (Sept 2016). doi:10.1109/TC.2015.2500572.

[23] A. Alhammad, R. Pellizzoni, Time-predictable execution of multithreaded applications on multicore systems, in: 2014 Design, Automation Test in Europe Conference Exhibition (DATE), 2014, pp. 1–6 (March 2014). doi:10.7873/DATE.2014.042.

[24] A. Alhammad, S. Wasly, R. Pellizzoni, Memory efficient global scheduling of real-time tasks, in: 21st IEEE Real-Time and Embedded Technology and Applications Symposium, 2015, pp. 285–296 (April 2015). doi:10.1109/RTAS.2015.7108452.

[25] P. Burgio, A. Marongiu, P. Valente, M. Bertogna, A memory-centric approach to enable timing-predictability within embedded many-core accelerators, in: 2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST), 2015, pp. 1–8 (Oct 2015). doi:10.1109/RTEST.2015.7369851.

[26] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, L. Sha, Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms, in: 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013, pp. 55–64 (April 2013). doi:10.1109/RTAS.2013.6531079.

[27] H. Yun, R. Mancuso, Z. P. Wu, R. Pellizzoni, Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms, in: 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014, pp. 155–166 (April 2014). doi:10.1109/RTAS.2014.6925999.

[28] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, T. Nolte, Contention-free execution of automotive applications on a clustered many-core platform, in: 2016 28th Euromicro Conference on Real-Time Systems (ECRTS), 2016, pp. 14–24 (July 2016). doi:10.1109/ECRTS.2016.14.