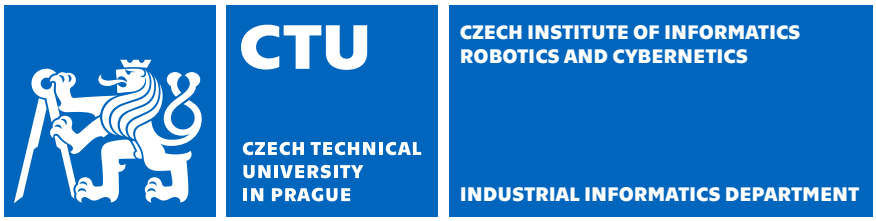


Combining PREM compilation and ILP scheduling for high-performance and predictable MPSoC execution

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

DOI: https://doi.org/10.1145/3178442.3178444
Cite as: J. Matějka, B. Forsberg, M. Sojka, Z. Hanzálek, L. Benini, and 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, pages 11–20, New York, NY, USA, 2018. ACM

Combining PREM compilation and ILP scheduling for high-performance and predictable MPSoC execution

Joel Matějka
Czech Technical University in Prague,
Faculty of Electrical Engineering
Prague, Czech Republic
matejjoe@fel.cvut.cz

Björn Forsberg
ETH Zurich,
Institut für Integrierte Systeme
Zürich, Switzerland
bjoernf@iis.ee.ethz.ch

Michal Sojka
Czech Technical University in Prague,
Czech Institute of Informatics,
Robotics and Cybernetics
Prague, Czech Republic
michal.sojka@cvut.cz

Zdeněk Hanzálek
Czech Technical University in Prague,
Czech Institute of Informatics,
Robotics and Cybernetics
Prague, Czech Republic
zdenek.hanzalek@cvut.cz

Luca Benini
ETH Zurich,
Institut für Integrierte Systeme
Zürich, Switzerland
lbenini@iis.ee.ethz.ch

Andrea Marongiu
University of Bologna
Bologna, Italy
a.marongiu@unibo.it

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 the memory bandwidth 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 an Integer Linear Programming scheduling model. Our compiler uses loop analysis and tiling to transform application code into PREM compliant binaries. 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-Systems-like scenario containing matrix multiplications and FFT computations on NVIDIA TX1. The results show that our approach maintains similar average performance and improves variance of completion times by a factor of 9.

CCS CONCEPTS

• **Computer systems organization → Real-time system architecture**; • **Software and its engineering → Scheduling**; **Compilers**;

PMAM'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5645-9/18/02…$15.00
https://doi.org/10.1145/3178442.3178444

KEYWORDS

PREM, predictability, LLVM, static scheduling, Integer Linear Programming, NVIDIA TX1

ACM Reference Format:
Joel Matějka, Björn Forsberg, Michal Sojka, Zdeněk Hanzálek, Luca Benini, and Andrea Marongiu. 2018. Combining PREM compilation and ILP scheduling for high-performance and predictable MPSoC execution. In *PMAM'18: 9th International Workshop on Programming Models and Applications for Multicores and Manycores, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178442.3178444

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 [15]) that guarantees *freedom from interference* can be employed.

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 [5].

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 [15], 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 [12], 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.

The scheduling tool introduced in this paper generates optimal schedules of PREM-compliant execution on multi-core CPUs. 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 formulate the scheduling algorithm as an Integer Linear Programming (ILP) problem. Although this does not allow solving large-scale problems, the ILP formulation gives a clear description of the problem in the form of a few equations and helps to classify the problem. Once we classify the problem, it is easier to find related problems and design efficient heuristics (such as [10]) capable of solving bigger instances. Also, solutions of the ILP can be used as a reference for comparison with heuristic algorithms on small instances.

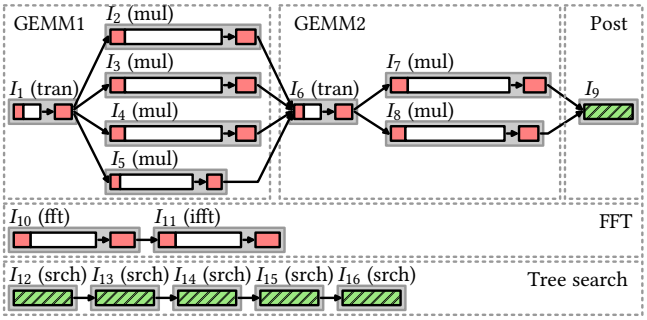


Figure 1: An example ADAS scenario.

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 algorithm in Section 4. Section 5 and Section 6 describe implementation and evaluation both the compiler and the scheduling on NVIDIA Tegra X1 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 and we then created several execution scenarios. One such scenario is depicted in Figure 1. The first algorithm is general matrix multiplication (GEMM), which is an essential operation in neural networks during forward propagation. 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.

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 [13] 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

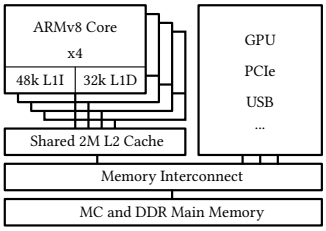


Figure 2: NVIDIA TX1 block diagram.

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 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.

3.1 PREM compiler design

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. Following this, the analysis passes (e.g., loop analysis) extract the required data from the source code, 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.

3.2 Compiler implementation

The following sections describe in greater detail the operations of the PREM compiler.

3.2.1 Preprocessing. As outlined in the previous section, the first step in the PREM compiler is to detect all memory objects that have

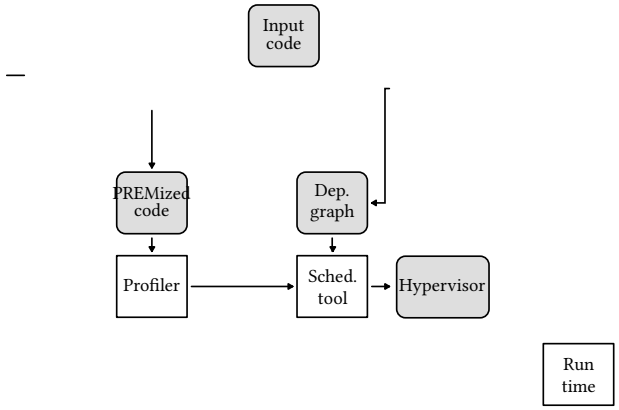


Figure 3: Block diagram of the toolchain proposed in this paper. Gray rounded boxes represent data, white rectangles are performed operations.

to be prefetched. In the first step, the memory objects are promoted to global variables to ensure that they are allocated statically or on the stack. This ensures that the data can be seamlessly prefetched and used at different points in the program.

3.2.2 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: possibility 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. 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.

The LLVM infrastructure provides the concept of *regions*, which are defined as single-entry single-exit 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. By mapping all uses of memory objects to the region in which they are used, the memory footprint of the region can be calculated and suitable PREM intervals selected based upon their footprint, as has been shown in [16].

For sequential code, no further steps are necessary. However, for places where the load or store happens within a loop, based on an address which is dependent on the loop iteration, further steps must be taken to calculate the range of addresses that are accessed. Thus, for regions that are within loops, *scalar evolution analysis* [9] can be employed to detect how variables change during the execution of the loop, to determine how large the memory footprint of the region is, based on the values that the loop-dependent addressing variable can take.

As loop dependent variables may span a wide range of addresses, it might not be possible to fit the entire iteration space within the local memory of the system. If this is the case, then the iteration space must be divided into smaller parts which do. This is achieved

through *tiling* [9], in which the iteration space is divided into multiple smaller parts (see groups of "mul" intervals in Figure 1). By selecting the amount of iteration that the tile takes, it is possible to divide a loop into small portions that fit into the local memory to make these new regions suitable as PREM intervals.

Regions are represented as trees, where the parent region of a child region encompasses both the child region and the surrounding code. Once the memory footprint is decided, it is possible to select PREM intervals by traversing the region tree from the root, selecting regions that fit in the local memory as PREM interval, such that every part of the code belongs to exactly one interval.

3.2.3 Transformation. Once PREM intervals have been selected through one of the two above methods, the transformation takes place. Thus, the region is outlined into a new function, and the function is cloned into three copies: one for prefetch, one for compute, and one for writeback, thus matching the PREM phases. These cloned functions are individually specialized to perform their intended task.

The prefetch function has all non-essential control flow and instructions removed, i.e., all instructions that are not required for the calculation of the addresses to load are discarded, including branches. The load operations themselves are replaced with prefetch instructions, and thus the prefetch function has been reduced to the minimum required to perform its task. The writeback function is transformed in the same way, except that cache flush & invalidate instructions are used in place of prefetches, such that the specific cache lines that have been loaded are written back to memory.

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 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.

4 SCHEDULING

After the code is generated by the compiler, we use the scheduling algorithm described in this section to optimally schedule parallel execution of the code on the multi-core target platform. The goal is to minimize completion time of the last executed interval, while simultaneously ensuring no interference at memory bus.

4.1 PREM application model

The application transformed into PREM compliant code has 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 scenario transformed into PREM compliant code is 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.

In order to improve flexibility of the scheduling algorithm, we allow the application to busy wait between the end of a compute

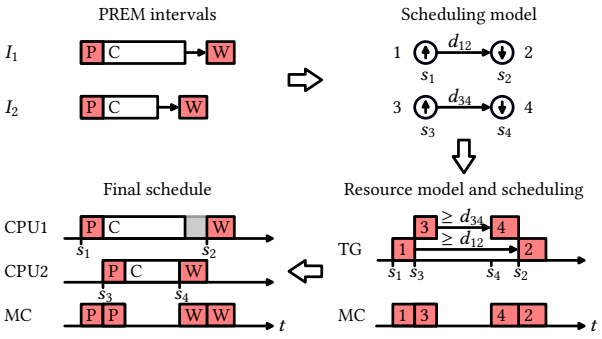


Figure 4: Translation of two PREM intervals into scheduling model and backwards

phase and start of the follow-up write-back phase (represented by arrows between compute and write-back phases), so that another core can complete its memory phase started during the compute phase of the first core.

For the sake of model simplicity, we consider compatible intervals as memory phases (requiring exclusive memory access). This approach can make the final schedule more pessimistic because some compatible intervals can utilize little memory bandwidth and simultaneous execution of multiple compatible intervals would not significantly affect their execution times. Alternative approach would be to allow scheduling of multiple compatible intervals at the same time. In such a case, the WCET estimation can be more difficult. We will pursue this approach in our future work.

For each phase of our model, we need to know its WCET. Compared to unrestricted execution models, determining WCET of PREM compliant code is pretty straightforward as the PREM model limits possible inter-core interference by ensuring exclusive access to the shared memory in prefetch, write-back and compatible phases and availability of all required data in local memory in compute phases. As demonstrated in our experimental evaluation, WCET times obtained by simple profiling match real execution times with sufficient accuracy.

4.2 Scheduling Model

To construct optimal application schedule, we execute our scheduling algorithm on the scheduling model derived from the PREM application model described above. We first describe the model conversion informally and follow with formal description.

We formulate the scheduling problem as a resource-constrained project scheduling problem (RCPSP) with multi-resource activities. We use a trivial example in Figure 4 to demonstrate conversion of a set of PREM intervals into our scheduling model consisting of *activities*. We also show how the final schedule is constructed from 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. 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.

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_1 and s_2 (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) [10]. The take-give resource may be seen as a counting semaphore with the capacity equal to a 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.

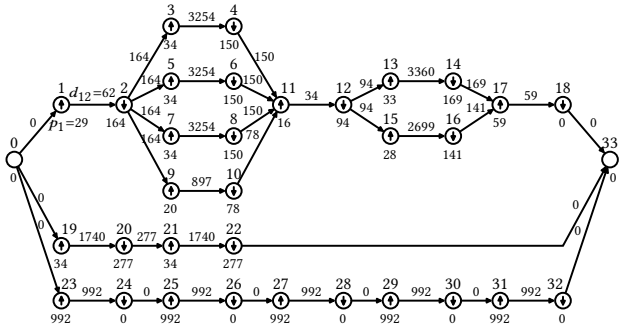
Our scheduling algorithm takes the activities as input and produces the schedule in which all resource requirements and temporal constraints are met. As our target platform is a symmetric multiprocessor system, it is not necessary to assign activities to particular cores. It is sufficient to determine 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 runtime 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 4).

We model compatible intervals similarly to predictable intervals with zero length of compute and write-back phases. Therefore, 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.

In the following, we describe our model in a more formal way. From a set of intervals, we create a project consisting of a set of $n+2$ non-preemptive activities $\mathcal{V} = \{0, 1, 2, ..., n+1\}$. Let $p_i \in \mathbb{R}_0^+$ be the execution time (sometimes called also processing time) of activity i and $s_i \in \mathbb{R}_0^+$ be the start time of activity i. Activities 0 and $n+1$ with $p_0 = p_{n+1} = 0$ denote "dummy" activities which represent the project beginning and the project termination, respectively. The activities correspond to nodes of the directed acyclic graph $G = (\mathcal{V}, \mathcal{E})$ where $\mathcal{E}$ is a set of edges representing temporal constraints between nodes. Each edge $e_{ij} \in \mathcal{E}$ from node i to node j is labeled by weight $d_{ij} \in \mathbb{R}_0^+$. The start times of activity i and activity j are subject to the *temporal constraint* given by inequality

$$s_j - s_i \geq d_{ij} \qquad \forall e_{ij} \in \mathcal{E}. \tag{1}$$

We model multi-core CPU as one take-give resource with capacity $Q \in \mathbb{Z}^+$ units (can be extended to multiple resources representing, e.g. another CPU cluster). Each occupation (i.e., interval between take and give activities) is linked with two activities, as indicated by $a_{il} \in \{0, 1\}$. The a_{il} parameter is equal to one if and only if occupation i starts its execution at s_i, the start time of the activity which takes a take-give resource, and finishes its execution



Take-give resource requirements: $a_{1,2} = 1$, $a_{3,4} = 1$, $a_{5,6} = 1$, $a_{7,8} = 1$, $a_{9,10} = 1$, $a_{11,12} = 1$, $a_{13,14} = 1$, $a_{15,16} = 1$, $a_{17,18} = 1$, $a_{19,20} = 1$, $a_{21,22} = 1$, $a_{23,24} = 1$, $a_{25,26} = 1$, $a_{27,28} = 1$, $a_{29,30} = 1$, $a_{31,32} = 1$ and the others $a_{il} = 0$

Figure 5: DAG constructed from the scenario in Figure 1

at $C_l = s_l + p_l$, completion time of the activity l which gives the take-give resource back (i.e., releases).

Figure 5 shows a directed acyclic graph (DAG) for the scenario shown in Figure 1. Each activity is represented by a node labeled by its id (the value above the node), its execution time p_i (the value below the node). Each edge is labeled by its start-to-start temporal constraint. All activities are executed on one resource with capacity one (i.e., a memory controller), and each occupation is executed on one unit of the take-give resource with capacity of four units (i.e., CPU cores at our platform). There is one occupation per each PREM interval given by a_{il} values in the legend of the figure. For example, $a_{1,2} = 1$ indicates that the core is taken at the start of activity 1 and released at the completion time of activity 2.

A feasible schedule (i.e., all time constraints are met) of this instance is shown on the Gantt chart in Figure 6. One can check that all the time constraints are met (e.g., activity 2 is scheduled 61 time units after the start of activity 1, i.e., the prefetch phase of length 29 and subsequent compute phase of length 33 have enough time to be executed), the resource constraints are met (e.g., when a activity requiring MC is executed, then no other activity requiring MC is scheduled at the same time) and the take-give resource constraints are met (the occupations do not use more than four CPU cores at any point in time).

A schedule is given by the activity start times s_i and variables $z_{iv}^{\sim}$ which denote assignment of occupations to take-give resources. The assignment $\tilde{z}_{iv} \in \{0, 1\}$ is equal to 1 if occupation i is assigned to unit v of the take-give resource, and 0 otherwise. Consequently, equation $\sum_{v=1}^{Q} \tilde{z}_{iv} = a_{il}$ holds for each $(i, l) \in \mathcal{V}^2$. A schedule $S = (s, z, \tilde{z})$ is feasible if it satisfies the temporal, resource and take-give resource constraints.

4.3 ILP formulation

To find an optimal schedule of the above defined scheduling model, we formulate the scheduling problem as an ILP problem. Figure 7 shows the complete ILP formulation. Let x_{ij} be a binary decision variable such that $x_{ij} = 1$ if activity i is followed by j in the schedule and $x_{ij} = 0$ if activity j is followed by i. Constraint (3) is a direct application of the temporal constraint (1) from Section 4.2. Constraints (4), (5) correspond to the resource constraints. UB denotes big positive number (e.g. upper bound of C_{MAX}). When $x_{ij} = 0$, constraints (4) and (5) reduce to $s_j + p_j \leq s_i$, i.e., j is followed by i.

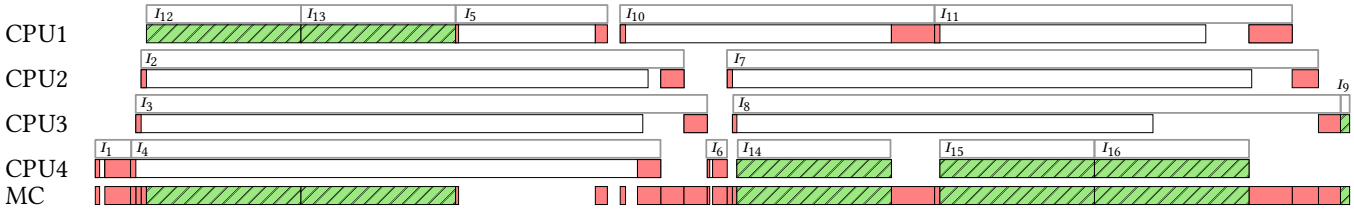


Figure 6: Gantt diagram with optimal schedule for the scenario in Figure 1

$$\min s_{n+1} \tag{2}$$

subject to:

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

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

$$s_i - s_j + UB \cdot x_{ij} \leq UB - p_i, \qquad \forall (i,j) \in \mathcal{V}^2 : \ i \neq j \tag{5}$$

$$\tilde{p}_i = s_l + p_l - s_i, \qquad \forall (i,l) \in \mathcal{V}^2 : \ a_{il} = 1 \tag{6}$$

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

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

$$-\tilde{x}_{ij} + \tilde{y}_{ij} \leq 0, \qquad \forall (i,j) \in \mathcal{V}^2 : \ i \neq j \tag{9}$$

$$\tilde{z}_{iv} + \tilde{z}_{jv} - 1 \leq 1 - \tilde{y}_{ij}, \qquad \forall (i,j,l,h) \in \mathcal{V}^4, \forall v \in \{1, \ldots, Q\} : \ i \neq j, \ a_{il} \cdot a_{jh} = 1 \tag{10}$$

$$\sum_{v=1}^{Q} \tilde{z}_{iv} = a_{il}, \qquad \forall (i,l) \in \mathcal{V}^2 : \ a_{il} = 1 \tag{11}$$

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

When $x_{ij} = 1$, constraints (4) and (5) reduce to $s_i + p_i \leq s_j$, i.e., i is followed by j.

The inequalities (6), (7), (8), (9), (10) and (11) stand for the take-give resource constraints. The variable $\tilde{x}_{ij}$ has the same meaning as x_{ij} for the resource constraints. The main difference is that $\tilde{p}_i$, the execution time of an occupation, is a variable while p_i, the execution time of an activity, is a constant. Execution time $\tilde{p}_i$, expressed in equation (6) is given by s_i, the start time of activity i which takes the take-give resource and completion time of activity l which gives back the take-give resource. Since the take-give resource has several units (i.e., four CPU cores), we need to add binary variable $\tilde{y}_{ij}$ which in combination with $\tilde{x}_{ij}$ distinguishes the mutual relation of occupations i and j. Their relation is expressed by constraints (7) and (8). There are three feasible combinations:

(1) When $\tilde{x}_{ij} = 0$ and $\tilde{y}_{ij} = 0$, constraints (7) and (8) reduce to $s_j + \tilde{p}_j \leq s_i$, i.e., j is followed by i.
(2) When $\tilde{x}_{ij} = 1$ and $\tilde{y}_{ij} = 0$, constraints (7) and (8) reduce to $s_i + \tilde{p}_i \leq s_j$, i.e., i is followed by j.
(3) When $\tilde{x}_{ij} = 1$ and $\tilde{y}_{ij} = 1$, constraints (7) and (8) are eliminated in effect and the occupations i and j must be scheduled on different units.
(4) Combination $\tilde{x}_{ij} = 0$ and $\tilde{y}_{ij} = 1$ is not feasible due to constraint (9).

The number of units is limited using variable $\tilde{z}_{iv}$ in constraints (10) and (11). Constraint (11) states that each occupation i is assigned to one unit of take-give resource. From constraint (10), it follows that when two occupations i and j can overlap, i.e., $\tilde{y}_{ij} = 1$ then the occupations cannot be processed on the same unit v since $\tilde{z}_{iv} + \tilde{z}_{jv} - 1 \leq 0$.

Finally, the objective function of the ILP model (2) minimizes the start time of the dummy activity $n + 1$, i.e., the last activity of the schedule.

5 IMPLEMENTATION AND LIMITATIONS

We use the LLVM compiler infrastructure [12] 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 restrictions to the supported C/C++ codes:

- the code cannot contain any form of recursion,
- all loops have to be bounded by constant value so that scalar evolution analysis can be used to analyze the loops,
- all variables have to be allocated statically or on the stack,

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 [3]. In light of this, these restrictions do not impose any severe limitations to real applications.

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. 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 [13], 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.

It has to be underlined that the mentioned problems are related to the relatively early stage of development of our tools, and not to an inherent limitation to the methodology. Most of these restrictions will be lifted as our technology gets more mature.

6 EXPERIMENTAL EVALUATION

In order to 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 the three ADAS kernels described in Section 2 (GEMM, FFT, and binary tree search). Such batches are instantiated with different parameters to create a number of use-case scenarios, as shown in Table 1. 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, solved the ILP problem, 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 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 of 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. Subsequently, the ILP model solved in IBM ILOG CPLEX Optimization Studio gives start times which define sequencing of memory intervals in our test bed.

We evaluate the effectiveness of the approach on four scenarios, each scenario having only 16 intervals, because of the time complexity of the ILP solution. Faster heuristic scheduling algorithms are planned for future work. Table 1 describes compositions of the scenarios. Each application of a scenario is described by two numbers – count and parallelism. We explain the meaning of the numbers on *Scn. 1*, which is the scenario from Figure 1. The first application is two subsequent GEMMs ($C = \alpha A \times B + \beta C$) where

Scenario	Scn. 1	Scn. 2	Scn. 3	Scn. 4
GEMM count	2	2	1	2
GEMM parallelism	4, 2	4, 2	7	4, 2
FFT count	2	2	2	4
FFT parallelism	1	1	1	2
Search count	5	5	5	3
Search parallelism	1	2	1	1

Table 1: Composition of selected scenarios

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 (GEMM count 2, parallelism 4 and 2). 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 scenarios are following:

(1) this scenario corresponds to Figure 1 and has just been described,
(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 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 so that FFT completely fits into core-local memory. Binary 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. We call these code *Legacy* in this section.

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 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 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 pool 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 threads whose number equals to maximum parallelism achievable by the application. The threads are scheduled by the Linux `SCHED_FIFO` scheduler and all have the same priority.

6.1 Experimental results

In Table 2 the measured worst-case execution times (WCET) and mean execution times are shown for each of the four scenarios, both for PREM and *Legacy* executions. Furthermore, the schedule

Scenario		Scn. 1	Scn. 2	Scn. 3	Scn. 4
PREM	C_{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
Legacy	WCET (ms)	9.75	11.27	11.09	10.79
	Mean (ms)	7.77	9.11	8.92	8.93
ILP solving time (s)		41	73	7908	60

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

completion time C_{MAX} calculated by the ILP solver is 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 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 6.

The measured execution times of all 100 000 runs of our scenarios are presented in logarithmic scale histograms in Figures 8a–8d. The PREM schedule completion time C_{MAX} is shown as a dashed black line.

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 3.8%) in comparison to *Legacy* executions (up to 52.4%). We calculate the variance P for PREM as $P = 100 \times (WCET_{\text{PREM}}/BCET_{\text{PREM}} - 1)$ where $WCET_{\text{PREM}}$ 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 8d 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 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%). We calculate the WCET difference as $LP = 100 \times (WCET_{\text{Legacy}}/WCET_{\text{PREM}} - 1)$. 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. 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.

	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

In addition, we evaluated the application execution under the presence of TCP-based network communication. As can be seen from Figure 9, *Legacy* executions experience huge interference caused mainly by preempting the application by packet processing in the Linux kernel.

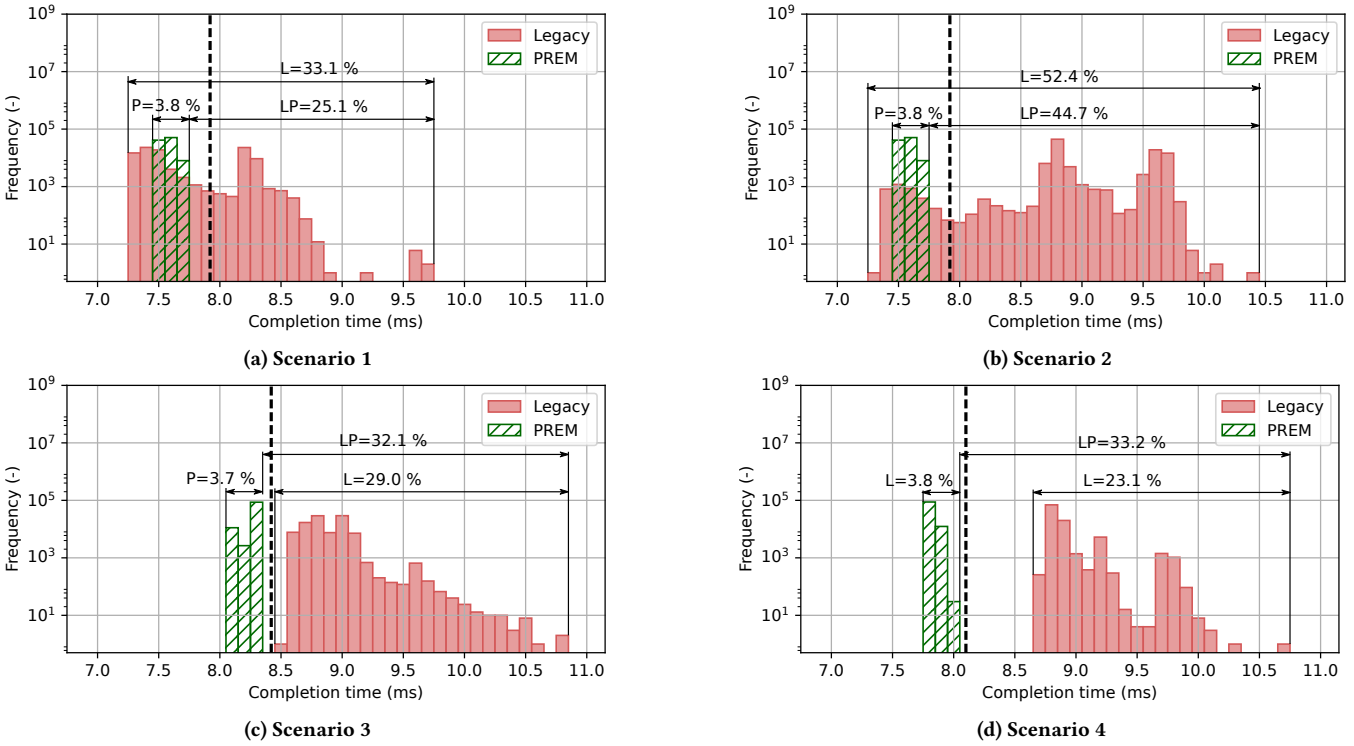
Table 3 shows the measured execution times and numbers of cache misses in Scenarios 1 and 2. Each predictable interval has measurements shown for each of the PREM phases (Prefetch, Compute and Write-back). For compatible intervals, the measured values are in the prefetch column only, as compatible intervals 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 an 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. [15]. The first attempt to extend PREM to multi-core systems was made by S. Bag



(a) Scenario 1

(b) Scenario 2

(c) Scenario 3

(d) Scenario 4

Figure 8: Histograms comparing completion times of scenarios with and without PREM applied

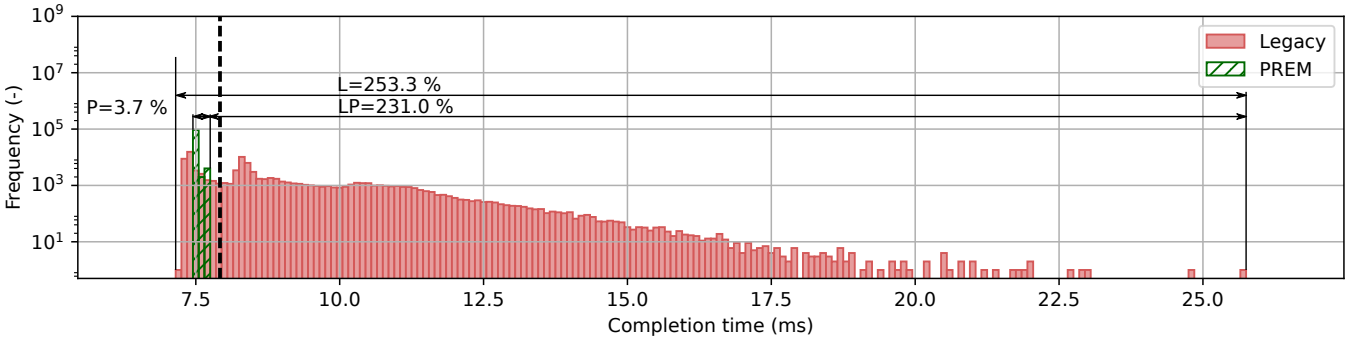


Figure 9: Histograms comparing completion times of scenario 1 with and without PREM applied and with external interference caused by running TCP communication (netcat) over 100 Mbps Ethernet

et al. [4]. 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. [17, 18] 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 [1] 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. [2] or Burgio et al. [7] 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 [15] 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. [14]. 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. [11] 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. [20] 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 [19]. 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. [6] 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 solving an integer linear programming model. Experimental evaluation shows that for the selected ADAS-like scenarios, PREM in combination with static scheduling brings the following benefits: i) Significant reduction of completion time jitter (max 3.8%) ii) WCET of the PREM schedule is always smaller than calculated schedule completion time and iii) 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%).

As our future work, we will evaluate the heuristic scheduling algorithm. We also aim at combining this work with PREM execution on the GPU.

ACKNOWLEDGMENTS

This work was supported by the HERCULES Project, funded by European Unions Horizon 2020 research and innovation program under grant agreement No. 688860. Collaboration on this paper was additionally supported through HiPEAC, a project that received funding from the European Union's H2020 research and innovation programme under grant agreement No 687698.

REFERENCES

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

[2] A. Alhammad, S. Wasly, and R. Pellizzoni. 2015. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 285–296. https://doi.org/10.1109/RTAS.2015.7108452

[3] Motor Industry Software Reliability Association and Motor Industry Software Reliability Association Staff. 2013. *MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems*. Motor Industry Research Association. https://books.google.ch/books?id=3yZKmwEACAAJ

[4] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo. 2012. 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*. 300–309. https://doi.org/10.1109/RTCSA.2012.48

[5] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. 2010. 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.

[6] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte. 2016. Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. 14–24. https://doi.org/10.1109/ECRTS.2016.14

[7] P. Burgio, A. Marongiu, P. Valente, and M. Bertogna. 2015. 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)*. 1–8. https://doi.org/10.1109/RTEST.2015.7369851

[8] Bjorn Forsberg, Andrea Marongiu, and Luca Benini. 2017. GPUguard: Towards Supporting a Predictable Execution Model for Heterogeneous SoC. In *DATE'17*.

[9] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly – Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 04 (2012). https://doi.org/10.1142/S0129626412500107 arXiv:http://www.worldscientific.com/doi/pdf/10.1142/S0129626412500107

[10] Zdeněk Hanzálek and Přemysl Šůcha. 2017. Time symmetry of resource constrained project scheduling with general temporal constraints and take-give resources. *Annals of Operations Research* 248, 1 (01 Jan 2017), 209–237. https://doi.org/10.1007/s10479-016-2184-6

[11] Konstantinos Koukos, Per Ekemark, Georgios Zacharopoulos, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. 2016. 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, 121–131. https://doi.org/10.1145/2892208.2892209

[12] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[13] J. Liedtke, H. Hartig, and M. Hohmuth. 1997. OS-controlled cache predictability for real-time systems. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*. 213–224. https://doi.org/10.1109/RTTAS.1997.601360

[14] R. Mancuso, R. Dudko, and M. Caccamo. 2014. 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*. 1–10. https://doi.org/10.1109/RTCSA.2014.6910515

[15] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. 2011. A Predictable Execution Model for COTS-Based Embedded Systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. 269–279. https://doi.org/10.1109/RTAS.2011.33

[16] Muhammad Refaat Soliman and Rodolfo Pellizzoni. 2017. WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Marko Bertogna (Ed.), Vol. 76. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 24:1–24:23. https://doi.org/10.4230/LIPIcs.ECRTS.2017.24

[17] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. 2012. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems* 48, 6 (01 Nov 2012), 681–715. https://doi.org/10.1007/s11241-012-9158-9

[18] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. 2016. Global Real-Time Memory-Centric Scheduling for Multicore Systems. *IEEE Trans. Comput.* 65, 9 (Sept 2016), 2739–2751. https://doi.org/10.1109/TC.2015.2500572

[19] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. 2014. 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)*. 155–166. https://doi.org/10.1109/RTAS.2014.6925999

[20] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. 2013. 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)*. 55–64. https://doi.org/10.1109/RTAS.2013.6531079