HePREM: A Predictable Execution Model for GPU-based Heterogeneous SoCs

Björn Forsberg*, Luca Benini*, *Fellow, IEEE*, Andrea Marongiu†, *Member, IEEE*

Abstract—The ever-increasing need for computational power in embedded devices has led to the adoption of heterogeneous SoCs combining a general purpose CPU with a data parallel accelerator. These systems rely on a shared main memory (DRAM), which makes them highly susceptible to memory interference. A promising software technique to counter such effects is the Predictable Execution Model (PREM). PREM ensures robustness to interference by separating programs into a sequence of memory and compute phases, and by enforcing a platform-level schedule where only a single processing subsystem is permitted to execute a memory phase at a time. This paper demonstrates for the first time how PREM can be applied to heterogeneous SoCs, based on a synchronization technique for memory isolation between CPU and GPU plus a compiler to transform GPU kernels into PREM-compliant codes. For compute bound GPU workloads sharing the DRAM bandwidth 50/50 with the CPU we guarantee near-zero timing varibility at a performance loss of just 59%, which is one to two orders of magnitude smaller than the worst case we see for unmodified programs under memory interference.

Index Terms—Real-time and embedded systems, Languages and compilers, Graphics processors, Memory management, Reliability, Runtime environments, Parallel systems

1 Introduction

In the last years both industry and academia have been working towards the goal of autonomous vehicles and systems [1]. This requires embedded computers capable of ensuring that the data needed for the vehicle to take decisions autonomously is computed with strict real-time guarantees, as failure to do so could cause accidents and damage to person or property. Autonomous systems rely on algorithms which require significant computing power. Bleeding-edge demonstrators for autonomous driving (AD) systems are capable of managing prompt, compute-intensive elaboration of sensor data by employing powerful in-trunk compute servers [2]. However, as of today such servers are extremely power-hungry, making them practically impossible to commercialize and calling for solutions based on orders-of-magnitude more energy-efficient embedded systems-on-chip [1].

In recent years, there has been a push towards heterogeneous SoCs for commercial off-the-shelf (COTS) embedded computing, which combine a general-purpose CPU with a programmable, data parallel accelerator such as a GPU [3] [4]. While these systems are capable of sustaining adequate GOps/W targets for the requirements of autonomous navigation workloads, their architectural design is optimized for best-effort performance, not at all for timing predictability. To allow for system scalability to hundreds of cores, resource sharing is a dominating paradigm at every level in these SoCs. In particular, it is commonplace to employ a globally shared main memory architecture between the CPU and GPU. This has large benefits in energy savings [5] due to reduced replication of power hungry hardware, and improves programmability, as programmers do not need to handle data movements between two discrete memories [6] when *offloading* computation to the accelerator. On the other hand, the execution of CPU and GPU programs becomes susceptible to *interference* from each other's accesses to main memory (and from other peripherals' accesses), with significant impact on the execution time of real-time tasks [7] [8].

Custom-designed hardware for real-time systems [9] [10], is not always a viable solution, as it generally lags severely behind in performance and cost compared to COTS systems, due to longer time-to-market and limited production volumes, which prevent access to the latest CMOS technology nodes. Therefore, software mechanisms that enable timing predictable execution on COTS hardware are of high interest. Certification authorities are in the process of defining software development guidelines aimed at enabling the long-awaited adoption of multi-core processors in safety-critical domains [11]. Here, the concept of *robustness to interference* is central, and achieved through strict time partitioning. As software partitions are guaranteed to execute in isolation, the worst-case execution times (WCET) of each partition can be computed/measured in isolation, greatly reducing the pessimism in traditional timing analysis.

The *predictable execution model* (PREM) [12] has been proposed as a solution to deliver *robustness to interference* in multi-core CPU systems sharing memory at various levels in the hierarchy. At its core, PREM enables timing predictable execution on non-predictable hardware by separating programs into intervals of *Memory* and *Compute phases*, which can be independently scheduled. Shared main memory is guaranteed to be only accessed during the *Memory Phase* and the system is scheduled such that only a single processing element executes a *Memory phase* at a time. By transforming the original program such that the *Memory phase* prefetches all required data into private memory, the *Compute phase* is able to operate without any external memory interference.

This paper presents a novel methodology to generalize PREM for timing-predictable execution on heterogeneous (CPU+GPU) COTS architectures with a single shared

- *B. Forsberg is with the Integrated Systems Laboratory, ETH Zürich, Zürich 8092, Switzerland. E-mail: bjoernf@iis.ee.ethz.ch.*
- *L. Benini is with the Electrical, Electronic, and Inforamtion Engineering Department of the University of Bologna, Bologna 40126, Italy, and with the Integrated Systems Laboratory, ETH Zürich, Zürich 8092, Switzerland. E-mail: lbenini@iis.ee.ethz.ch.*
- *A. Marongiu is with the Department of Physics, Informatics and Mathematics of the University of Modena and Reggio Emilia, Modena, Italy. Email: andrea.marongiu@unimore.it*

DRAM. To simplify programmability, we build upon recent proposals for directive-based programming models [13], [14], [15] – which are more abstract than the low-level, involved coding style of CUDA or OpenCL – and apply the required transformations transparently to the application developer, as part of the compilation process. Specifically, we design PREM support on top of OpenMP [13], with an extended offloading runtime that interacts with a synchronization library – called *GPUguard* – that orchestrates the memory accesses between the CPU and the GPU.

Initial explorations have been presented in [16], while we make the following contributions:

- We describe novel compiler transformations to create optimized and independently schedulable PREM memory and compute phases, the key building block to achieve *robustness to interference* without having to idle the GPU when it cannot access the shared memory;
- We describe a software-controlled system-level memory scheduling infrastructure for predictable execution on heterogeneous SoCs, and extensively document the performance impact of memory scheduling;
- We present a full implementation of the proposed techniques based on the LLVM compiler infrastructure and the Linux kernel;
- We present an extensive evaluation of the proposed techniques, running on an NVIDIA Tegra X1 and providing detailed insight on their effect on timing predictability and performance.

The experimental results show that our solution is able to limit the execution time variance of GPU kernels to *near-zero* (max 3.5%), fully achieving the desired *robustness to interference* effect. For compute bound GPU workloads sharing the DRAM bandwidth 50/50 with the CPU this is achieved at an average performance loss of 59%, with respect to an interference-free theoretical best-case execution.

The paper is organized as follows: Sec. 2 discusses basic notions and assumptions. Sec. 3 describes our Compiler/Runtime support and Sec. 4 its evaluation in terms of predictability and performance. Sec. 5 presents related work, and Sec. 6 concludes.

2 Background

2.1 Architectural Template and Hardware Platform

As an architectural template we consider a System-on-Chip (SoC) that consists of a CPU and a GPU complex, sharing a single DRAM (the *global memory*). This means that all data transfer occurs on-chip, and is subject to memory interference. We consider a multi-core CPU, with at least one level of private caches. A miss in the last-level cache (LLC) is redirected to the main memory controller. The GPU complex consists of many simple cores grouped in *clusters*. Within a cluster each core has access to a private L1 cache, as well as a shared software-managed scratchpad memory (SPM). The GPU complex may consist of any number of such clusters. Several clusters share a hardware-managed L2 cache. A miss in this cache is also redirected to the main memory controller, such that cache misses from different complexes may interfere with each other, leading to delays that are difficult to predict. In practice, most commercial systems adhere to this template [4], [17].

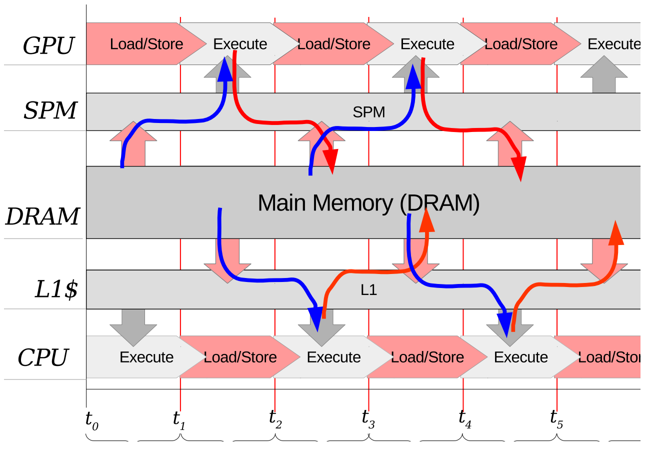


Fig. 1: A schematic view of how the memory separation property of PREM is enforced.

2.2 The Predictable Execution Model (PREM)

The Predictable Execution Model (PREM) was originally proposed in the context of single-core CPUs [18], to provide *robustness to interference* from peripheral (I/O) devices sharing the main memory. The concept was later extended to counter inter-core interference in multi-core CPUs [12]. PREM [18] separates programs in scheduling intervals that can represent memory or compute phases. By scheduling the system such that only a single actor is executing a memory phase at a time, PREM ensures that this memory phase will not experience any interference. As a consequence, the WCET of each phase can be calculated or measured in isolation, leading to system composability and greatly reduced pessimism in the timing analysis.

A PREM interval maps to a region of code whose memory footprint is small enough to fit into the local memory, such as a private cache or SPM. Within the interval, the execution is split into three phases, *Load*, *Execute*, and *Store* (LES), where only the *Load* and *Store* phases access the global memory, transfering the required data to/from the local memory. These two phases together make up the PREM *Memory phase*. The *Execute* phase, which coincides with the PREM *Compute* phase, can then operate on the local data without accessing the shared memory. Separating memory and compute phases enables the execution to continue even when the program does not have access to the global memory, minimizing the time the task would otherwise idle.

Previous work on PREM has been focused on the CPU, where PREM can be enforced at runtime by scheduling the Memory and Compute phases individually using the system scheduler. On heterogeneous systems, the problem becomes more involved, as a) the PREM schedule must be enforced within both the CPU and the accelerators locally, and b) between all the CPU and accelerators in the system globally. Fig. 1 shows a breakdown of a CPU+GPU heterogenous PREM system in discrete time steps. In the interval $[t_0, t_1)$ the GPU is permitted to access DRAM, and uses this time to load data from the DRAM to the local memory (in this case the SPM). Meanwhile, the CPU is not permitted to access DRAM, but is able to continue execution by using data in its local memory (e.g, L1 cache). In the next time step $[t_1, t_2)$ the roles are exchanged, and the GPU is working on the local data it fetched during the previous time step, while the CPU is permitted to load/store new data to/from its local memory.

To ensure CPU and GPU memory transfers do not interfere with each other, we use a time-triggered global arbitration mechanism, that can be implemented with timer interrupts. At these points, the two devices synchronize with each other to determine which device can access memory during the next *period*. The frequency at which these timer interrupts happen can impact the execution time of the CPU and GPU programs negatively if they are not well aligned to the actual execution time of the workload, as they would idle until the synchronization occurs. The optimization of these frequencies is out of the scope of this paper, but we provide insights on this through the use of two different timer configurations: *best fit* and *fair*. In the former, the timers are perfectly aligned with the execution time of the GPU program, and in the latter they are set such that CPU and GPU memory time allocations are equal.

3 HePREM Design and Implementation

3.1 HePREM Design Decisions

HePREM is based on five key design decisions:

Real-time coding standards – As the target of PREM is the enabling of real-time execution, we focus our efforts towards code written in accordance with best practices for real-time systems. Such best practices have been designed to enable certification of safety critical software in diverse domains, such as the MISRA guidelines for the Automotive industry [19]. One of the main benefits of code written in accordance with such guidelines is that it is subjectible to static analysis, i.e., program behavior can be determined at compilation time. For HePREM, this means that exact analysis tools can be used in place of, e.g., profile-based techniques.

High Level Language Compilation – The push towards the adoption of directive-based programming models such as OpenMP and OpenACC in the context of GPGPUs is constantly growing, as traditional languages like OpenCL and CUDA offer a low-level programming style. OpenMP ensures that also the non-expert user can easily code the desired functionality by just abstractly indicating which loops are to be offloaded to the accelerator[1]. Not only is this *per-se* a valuable feature worth to preserve, it also gives the compiler the freedom to determine the best work partitioning and data movements for predictability, without conflicting with such low-level decisions made by the programmer.

We focus on the subset of OpenMP directives that are suitable for execution on GPUs, i.e., Single-Instruction Multiple-Data (SIMD) execution. These are expressed using `parallel for` loops, that are distributed over the GPU clusters using OpenMP *teams*. Based on the findings of [20], we focus our efforts on statically scheduled loops, as these perform significantly better than dynamic scheduling on GPUs[2], and achieve similar performance as native CUDA.

PREM Enforcement Granularity – Predictability can be enforced at different granularities in the system, most prominently at offload boundaries, or *continously* throughout the execution of GPU kernels. However, implementing PREM synchronization at the boundaries of a whole offloaded task has, in our opinion, several drawbacks.

1. Advanced directives provide detailed control to expert users.

2. HePREM also supports reductions, tuning of *num_threads* and *num_teams* (corresponding to CUDA *blockDim* and *gridDim*), and other OpenMP clauses that can be applied to parallel for loops.

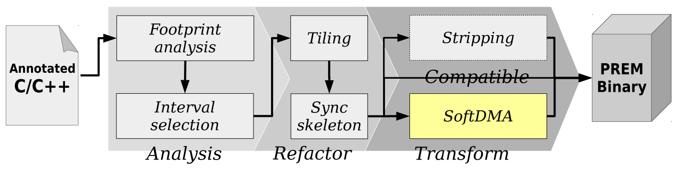


Fig. 2: The analysis and transformation steps taken as part of the PREM-enabling compiler passes.

First, it is not possible to control the granularity of the PREM phases without transforming the kernel code, either by inserting data movements within the kernel, or by splitting the program into multiple kernels and orchestrating data-movements on a per-offload basis. Controlling the granularity is key, as GPU kernels could otherwise block the CPU such that it is starved for memory. Second, the local memory of the GPU is cleared at the end of an offload, and not addressable from the host, i.e., data can not be copied into the scratchpad using the `memcpy` functionality. Thus, even if the program is split into multiple kernels, the data movements would still be performed by the kernel code itself. Third, splitting the kernels would incur further offloading overhead, as the GPU would have to be reconfigured for every offload. This operation is also part of the closed-source driver, and can not be relied on for timing predictable execution, unless given by the manufacturer.

In contrast, PREM intervals can be easily created from loop structures in the kernel code itself, through the process of *tiling*. By treating a loop iteration as an atomic unit to construct PREM intervals, they can be grouped together into blocks that are sized to perfectly match the local storage. In light of this, we argue that the most reasonable granularity to enforce predictability is *continously* over the kernel execution, by explicitly encoding the PREM phases and synchronization points within the GPU kernel[3].

Staging data through the Scratchpad – PREM can be implemented at any level of the memory hierarchy. In the provided architectural template, there are two main options available for the GPU, the SPM (CUDA *shared memory*) and the hardware-managed last-level cache (LLC). While it has been shown [21] that GPU caches could in theory be used for PREM, it has also been shown that kernels need to be well engineered to ensure a cache-friendly access patterns. Even after this, the random replacement policy is still a great hurdle to guarantee that data is locally available. As the main goal of the presented work is to achieve predictability, we have opted for the software managed SPM, that is not subject to un-controllable hardware eviction policies.

CPU+GPU Synchronization – To exchange the memory access token between the CPU and the GPU, explicit synchronization is required. However, as no current generation GPU system include any hardware-supported way to do this, we have decided to synchronize over shared memory. This is a portable approach available on any platform, and does not require any particular hardware support.

3.2 HePREM Compiler

The compilation flow consists of a three-phase process, as presented in Fig. 2. The *Analysis* phase devises a tiling factor

3. The offloading point also requires protection to avoid interference due to OS scheduling jitter, i.e., interference from other tasks competing for processor time, but this no longer requires access to the proprietary drivers. This is further explained in Sec. 3.3.2.

```
for(int i = 1; i < 500; i++) {
  A[i] = i;
}
```

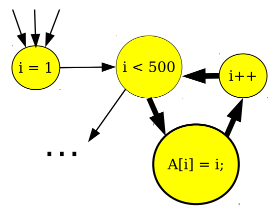


Fig. 3: Original loop

for the loops such that the memory footprint of each tile fits into the local memory. The *Refactoring* phase applies loop tiling based on the selected tiling granularity and inserts a synchronization skeleton to enforce the PREM isolation property. The *Transformation* phase specializes the original code to generate individual PREM phases. This section describes these steps in further detail. To understand how the compiler operates, it is useful to have an understanding of the *control flow graph* (CFG) of the code as it goes through the different transformation stages of the compilation process. Consider Fig. 3 that shows both C code and the CFG of a simple loop. In this representation, the loop is defined by an initial *assignment*, followed by the *loop condition*, and finally the *increment*. The code executed within the loop is referred to as the loop *body*. Within the compiler, this code is represented as a directed graph of *basic blocks*, where a basic block is a sequence of instructions where the terminating instruction is the only *branch* instruction. The destination of the terminating *branch* instructions of each *basic block* provide the edges between the *basic blocks* of the CFG.

3.2.1 Analysis

The first Analysis step is to identify code regions that can be turned into PREM intervals, and determine their memory footprint such that the largest possible regions are selected. As OpenMP offloading is based on loops, the footprint analysis focuses on determining the memory footprint of loops. For this purpose, Scalar Evolution (SCEV) [22] analysis is employed (built into LLVM), which is able to determine how the loop *induction variable (IV)* changes over the execution of the loop. SCEV analysis requires loops to be statically analyzable, i.e., the iteration space of the loop must not depend on any information that is not available at compile time. This is in line with typical requirements on real-time code (see Sec. 3.1), and allows the analysis to always return exact results[4]. By combining this information with the *loop variant* data accesses[5] performed within the loop it is possible to determine the memory footprint of each loop. Algorithm 1 shows how the range of addresses M accessed is determined. Lines 6-7 use SCEV to determine the addresses accessed by loop-variant memory operations s_A, based on the initial value of the IV (start), its increase over successive iterations (step) and the total number of iterations (tripcount). For loop invariant accesses, line 10 records the single address loaded or stored. Lastly, on lines 13-15 the memory accesses of sub-loops in loop nests are analyzed recursively, and at the end of the recursion, the memory footprint is determined from the addresses accessed M.

4. The presented methodology does not fundamentally depend on any feature specific to Scalar Evolution, and other loop analysis techniques, including profile-based techniques or bounds provided through programmer annotations, could be used in their stead.

5. In a loop variant access, the address loaded or stored is different for each loop iteration, e.g., when iterating over an array.

Algorithm 1 Pseudo-code for memory footprint analysis.

```
 1: Input: Loop $L$
 2:   $A$ is a memory access in $L$
 3:   $s_A$ is a tuple describing the SCEV of $A$ in $L$ $(start, step, tripcount)$
 4: Output: Memory access map $M_L$
 5: for all memory access $A$ in $L$ do
 6:     if $A$.loopvariant($L$) then
 7:         $s_A$ = ScalarEvolution($A$, $L$)
 8:         $M$.addAddressRange($start = s_A.start$, $end = s_A.start + s_A.tripcount \times s_A.step$)
 9:     else
10:         $M$.addAddress($A$)
11:     end if
12: end for
13: for all Sub Loop $SL$ in $L$ do
14:     Recurse on $SL$
15: end for
```

Algorithm 2 Pseudo-code for the tiling decision.

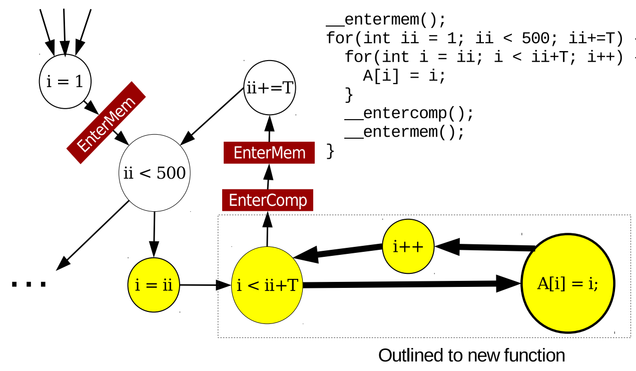
```
Require: Memory Footprint for all Loops, given by $Footprint()$, and calculated from $M$ in Algorithm 1.
Require: $S_{SPM}$ is the size of the local scratchpad
 1: if Loop is SIMT Loop then
 2:     if Footprint(Block iteration of Loop) $< S_{SPM}$ then
 3:         Set $blockDim$ to largest multiple of $num\_threads$ such that Footprint($Tile_{blockDim}$) $< S_{SPM}$
 4:     else
 5:         Failure
 6:     end if
 7: else
 8:     Set $blockDim$ to $num\_threads$
 9:     for all Sub Loop in Loop $\cup$ Sub Loops do
10:         Set $tileDim_{SubLoop}$ to largest value $V$ such that Footprint($Tile_{blockDim \times tileDim}$) $< S_{SPM}$
11:     end for
12:     if $V < 1$ then Failure end if
13: end if
```

3.2.2 Refactoring

Once the memory footprint has been calculated, PREM intervals are selected by loop tiling, after which a synchronization skeleton separating the PREM phases is inserted.

Tiling - Algorithm 2 shows how the memory footprint is used to define the granularity of PREM intervals, by selecting tiles that fit within the size S_{SPM} of the SPM. Lines 1-6 handle the case when the tiling is performed on the outermost loop in the annotated *loop nest*. We refer to this outermost loop as the *SIMT Loop* (SIMT = Single Instruction Multiple Threads), as work distribution among GPU threads is determined by the iteration space of the outermost loop in OpenMP. We refer to the tile dimension influenced by the *SIMT Loop* as the $blockDim$, and it is constrained to be a positive multiple of the OpenMP *num_threads* annotation, as an uneven distribution of the *SIMT loop* iterations cause some threads to have no work to perform as part of the *tile*, reducing performance. Therefore, we define a *block iteration* of the *SIMT loop* as each thread executing exactly one iteration in parallel. We select the largest such multiple that produces a tile that fits in the SPM (line 3). If no such tile can be created, the algorithm returns a *failure*[6] (line 5), however, none of the kernels we will show Sec. 4 trigger this case. Lines 7-13 of Algorithm 2 handle the tiling of N levels of inner loops. Also in this case, we respect the positive multiple constraint on the $blockDim$ (line 8) to ensure a balanced workload. The iteration spaces of the remaining N dimensions of the tile, which we refer to as $tileDim_n$

6. It would be possible to create tiles where some threads idle, but we opted for a compiler warning. This allows the OpenMP *num_threads* to be adjusted to enable balanced tiles and better performance.



```
__entermem();
for(int ii = 1; ii < 500; ii+=T) {
  for(int i = ii; i < ii+T; i++) {
    A[i] = i;
  }
  __entercomp();
  __entermem();
}
```

Fig. 4: Tiled loop with *EnterMem*/*EnterComp* sync skeleton.

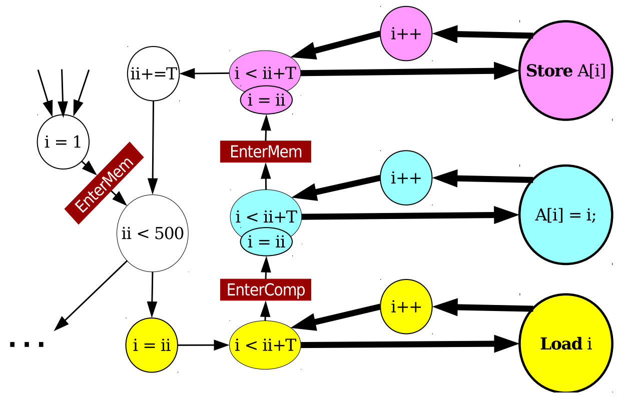


Fig. 5: Skeleton extended to Load Execute Store phases, with *EnterComp* and *EnterMem* synchronizations.

($n = 1 \dots N$), are local to each thread, and can be tiled freely. Thus we select the largest $tileDim$ possible V in accordance with the size of the local memory, S_{SPM} (line 10). If it is not possible to find V such that the loop fits into S_{SPM}, we use $V = 1$ if the loop has subloops, as we can then find a tiling granularity for the subloops on lines 9-11. If there are no subloops (i.e., innermost loop) and we cannot tile it to be smaller than S_{SPM}, setting $V = 0$ triggers a Failure on line 12[7]. Ultimately, the inequality in Equation 1 must hold, meaning that the more threads that are used per cluster ($blockDim$), the smaller the $tileDim$ (iterations local to the thread) will become.

$$S_{SPM} \geq blockDim \times tileDim = blockDim \times \prod_{n=1}^{N} tileDim_n \tag{1}$$

The tiling of the loop adds the uncolored nodes in the CFG, and the inner loop in the C code, shown in Fig. 4. Note that our approach is not specific of any tiling technique. We plan to investigate as future work the use of more advanced methods [23] [24]. After the loop has been tiled, the tile itself is *outlined* to a new function (this is the opposite of *inlining*), as shown by the dotted rectangle in Fig. 4.

Synchronization skeleton - Following this, the resulting tiles are separated through the insertions of synchronization calls into the predictable runtime, to be presented in Sec. 3.3. Through these synchronization points, the GPU program is effectively divided into PREM memory and compute phases, as shown in Fig. 4 by the red squares in the CFG, representing the function calls in the code. Thus, every instruction following a synchronization is part of the following PREM phase, as specified in the call, up until the next synchronization. The next section describes how the PREM phases are specialized to perform the required Load, Execute, and Store operations, as described in Sec. 2.2.

3.2.3 Transformation

To separate memory accesses from computation, the code must be divided into three distinct steps that are *specialized* to perform the *Load*, *Execute*, and *Store* (LES) steps of the PREM intervals. Naively, the PREM phases can be created by cloning the tiled loop into two additional copies, and placing one after each synchronization, as shown in Fig. 5. The Load and Store phases are then created simply by *stripping* the non-load/store code out of the respective phases,

7. Addressable by splitting the inner loop into two PREM intervals.

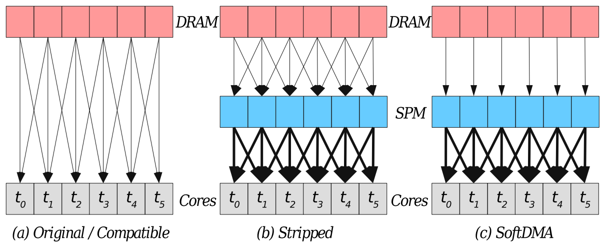


Fig. 6: The memory access patterns of the original program, and the PREM memory phases created with the *stripping* and *SoftDMA* techniques.

and the execute phase restructured to use the intermediate load/stores from these phases. The *stripping* technique is similar to Decoupled Access Execute [25] that has been used successfully on the CPU. The state of the art for PREM on GPU [16] uses this technique (with limited success), and we will revisit it in the evaluation in Sec. 4.

SoftDMA – However, for the code to perform well on the GPU, where hundreds or even thousands of threads are executing the same code, additional care must be taken when creating the Load and Store phases. The performance of GPU programs heavily depends of regular and well-organized memory accesses. In this sense, the *stripping* technique has two main disadvantages. First, reusing the original control flow means that multiple threads may load/store the same data to the SPM, leading to less effective use of the memory bandwidth. Second, the strict adherence to the original control flow means that sub-optimal access patterns may be inherited from the compute patterns, that could have been optimized if the memory accesses were decoupled from the point of use [26]. Algorithm 1 provides the compiler with the necessary information to create better optimized memory phases that lift these limitations, through a novel technique we refer to as *SoftDMA*.

As a motivating example, consider a kernel that computes *A[i-1] + A[i] + A[i+1]*, i.e., the thread executing the ith iteration will access the same memory as the threads executing the $i-1$th and $i+1$th iterations, as shown in Fig. 6a. In the original program this is required, as each thread fetches its data from the global memory. However, if *stripping* is used, this duplication will unneccessarily be inherited by the memory phase, as shown in Fig. 6b. In contrast, *SoftDMA* enables each unique accesses to be mapped to a single thread, as shown in Fig. 6c.

Instead of reusing the original control flow for the Load and Store phases, as shown in Fig. 5, optimized *SoftDMA* memory phases are created as loops in Algoritm 3.

To present the algorithm we refer to 2D structures, but the algorithm generalizes to data of any shape. Access code is created for each data structure D in the access map M (line 1), handling both cases identified as part of the footprint analysis: loop variant accesses to *composite types* (arrays of any dimension, and structs) on lines 2-11, and loop invariant accesses, e.g., scalars, on lines 13-16. For the latter, no loops are needed, and the data is loaded as is (line 14). The helper function *createLoop(start, stop, step)* is used to create a loop on the form `for(`*iv := start*; *iv < stop*; *iv += step*`)`. $L.iv$ is the IV (Sec. 3.2.1) of the created loop L.

For loop variant accesses, the algorithm operates on a per-dimension basis, where each dimension represents one layer of pointer dereference. For example, an access to a 2D structure A[i][j] requires three dereferences: The first to identify the location of A in memory, the second to identify the offset to the row i, and the last to identify the column j (offset within the row). We enumerate these dereferences in $derefChains$, starting from the base structure and handle them one by one (line 3). Important to realize is that only the last dereference will address a sequential piece of memory, as only rows are laid out sequentially – traversing a column implies an access pattern with strides the length of a row.

To achieve coalesced memory accesses, we therefore assign these accesses to neighboring threads: Line 5 creates a loop indexed by the *threadIdx*, thus mapping the accesses over individual threads such that all threads will access data as close to each other as possible. On line 6, we use this index to fetch the correct data from memory, using the offset between the *threadIdx* and the accessed data element, and the steps to recompute the address. For all other dimensions dereferenced, we create sequential loops within each thread (lines 8-9), that account for the offsets to the non-sequential memory ranges (e.g., row-by-row accesses in a 2D structure). We thus replace the suboptimal access behavior in $derefChain$ with new chains $derefChains'$ that are efficiently mapped to the iteration space of L, which are *pushed* into the loop body of L. By mapping the iteration space of the new loop L to the $threadIdx$, SoftDMA improves performance compared to *stripping* by ensuring that memory accesses in sequential memory are loaded in a coalesced manner, and that each element is loaded exactly once, thus adding the least possible amount of instructions to create the memory phases. As synchronization is implicitly provided by PREM, no additional synchronization is required.

Certain dereferences, such as A[B[i]], can not be known at compile time, as the value returned by B[i] can not be determined. For such cases, it is not possible for SoftDMA to coalesce memory accesses into A. However, SoftDMA is still able to prune duplicate accesses into B[i], limiting unneccessary transfers. Due to the sensitivity to memory access patterns, such constructs are commonly avoided in GPU code, and we do not further optimize for this case[8].

The use of *SoftDMA* only affects the Memory phases, i.e., the Load and Store transformations. The only transformation done to the Execute phase is to replace all accesses to global memory with accesses to scratchpad buffers, to/from which the Load and Store phases load/store data.

8. Instead, the *stripping* technique is used as fallback solution, if analysis provides insufficient information for SoftDMA code generation.

Algorithm 3 Pseudo-code for the SoftDMA decision.

```
Require: Memory Footprint given by $Footprint()$, and calculated from
    $M$ in Algorithm 1.
 1: for all data structures $D$ in $M$ do
 2:   if D.$derefChains.isLoopVariant$ then
 3:     for all dimension $d$ in $D.derefChains$ do
 4:       if $d.isSequential$ then
 5:         $L = createLoop(threadIdx, d.end \div d.step, blockDim)$
 6:         $D.derefChains'.push(L.iv \times d.step + d.start)$
 7:       else
 8:         $L = createLoop(d.start, d.end, d.step)$
 9:         $D.derefChains'.push(L.iv)$
10:       end if
11:     end for
12:   else
13:     for all dimension $d$ in $D.derefChains$ do
14:       $D.derefChains'.push(d)$
15:     end for
16:   end if
17: end for
```

3.3 HePREM Runtime: GPUguard

To orchestrate the memory accesses between the CPU and the GPU on a system level we introduce a runtime infrastructure for CPU-GPU synchronization and memory isolation, namely GPUguard. GPUguard answers calls to the EnterMem and EnterComp functions inserted into the transformed code as part of the compiler transformations.

3.3.1 System Co-scheduling through Synchronization

As the GPU and the CPU can only communicate through global memory, the PREM co-scheduling is performed through a synchronization flag, kept in the shared DRAM. At the end of each Memory or Compute phase, a synchronization takes place (as illustrated by the CFGs in Fig. 5), during which the memory *token* is exchanged between the CPU and the GPU, and dictates which of the two is allowed to access the global memory until the next synchronization.

The synchronizations are enforced through timer interrupts triggered on the CPU at the end of the WCET of each PREM phase, by setting a timeout at the start of each PREM phase. Thus, the length of each phase, $T_{compute}$ and T_{memory} respectively, must be programmed into the system so that the exchange of the memory token is correctly performed at the end of each phase. At the system level we only consider PREM Memory and Compute phases. Thus, each kernel has only two timer values associated with it, $E_{compute}$ and E_{memory}, as previously shown in Fig. 1. The system is scheduled such that the GPU has access to the global memory p_{memory} percent of the time, and conversely is not allowed to access memory $p_{compute}$ percent of the time, such that $p_{memory} + p_{compute} = 100\%$. As the enforced phase lengths E_{memory} and $E_{compute}$ of the system must be sized such that they both fulfil the system schedule parameters p, and be large enough to enclose the full phase times $E \geq T$, the phases are calculated as shown in Equation 2.

$$E_{memory} = \begin{cases} T_{memory}, & \text{if } \frac{T_{compute}}{T_{memory}} \leq \frac{p_{compute}}{p_{memory}} \\ \frac{p_{memory}}{p_{compute}} \times T_{compute} & \text{otherwise} \end{cases}$$
$$E_{compute} = \begin{cases} \frac{p_{compute}}{p_{memory}} \times T_{memory} & \text{if } \frac{T_{compute}}{T_{memory}} \leq \frac{p_{compute}}{p_{memory}} \\ T_{compute} & \text{otherwise} \end{cases} \quad (2)$$

Thus, the execution time of a tile, with the system schedule taken into account is $E = E_{memory} + E_{compute}$. The idle time I introduced into the the transformed kernel through the system schedule, can be determined as shown in Equation 3. When the schedule is created with E such that $I = 0$, we say that we have a *best-case* schedule. Note however, that as E is the worst case execution time, the system might still idle when the execution time is lower.

$$I = (E_{memory} + E_{compute}) - (T_{memory} + T_{compute}) \quad (3)$$

The timer interrupt handler implements the synchronization protocol, and is loaded into the kernel as a loadable module (LKM). Each time the GPU reaches a synchronization point, i.e., it wants to enter the next PREM phase, it writes a synchronization flag into the shared DRAM. Once the WCET for the PREM phase has expired, the timer expires and the handler in the LKM is invoked to perform the handover of the memory token. Synchronization is performed twice per PREM interval. Thus, taking the synchronization cost into account, the overall execution time estimate of each tile is $L_{interval} = E_{memory} + E_{compute} + 2 \times S$. As can be seen by inspection, the relative impact on the execution time of the synchronization is dependent on the execution time of the individual phases. If $E \gg S$ the synchronization cost will be negligible, but if $E \ll S$ it will dominate the overall execution time. A more in-depth discussion on this effect follows in Sec. 4.2.1.

3.3.2 Enforcement of CPU Memory Inactivity

We conservatively assume that the CPU tasks do not fully comply to PREM, and thus a separate mechanism must be employed to ensure that the CPU does not access global memory during the GPU memory phases. For this purpose *throttle threads* [27] are employed on the CPU. Throttle threads are executed at high-priority, and preempt any running task. While scheduled, the throttle thread itself only idles, and thus does not generate any memory accesses itself. We schedule CPU throttle threads during GPU memory phases, giving the GPU exclusive use of memory, which are descheduled when the GPU enters the compute phase.

Interference within the system consists of *memory interference* on the shared memory system, and *OS scheduling interference* on the CPU. The scheduling interference occurs when the Linux scheduler selects an interfering task from the ready queue in favor of the GPU offloading task, which may inject large delays (jitter) into the response time of the offloaded kernel [28]. To protect the offloading point from such jitter, extra synchronizations are inserted around the offload in the OpenMP runtime. Scheduling *throttle threads* during this critical point ensures that mutual exclusion of shared resources is achieved internally on the CPU.

In systems with PREM-compliant CPU tasks (enforced by a PREM scheduler), the *throttle thread* is not necessary, except for misbehaving tasks. As the architectural template (Sec. 2.1) enables PREM data staging through CPU caches, CPU memory management can be achieved as outlined in Fig. 1. This generalization is a key point of our future work, but out of scope in this exploration of PREM on the GPU.

4 Experimental Evaluation

Our experiments aim at assessing the performance and reduction in sensitivity to memory interference and scheduling jitter for the novel SoftDMA (henceforth SDMA) PREM kernels, comparing it to the unmodified baseline, and two previously proposed techniques: Compatible Intervals, and Decoupled Access Execute.

Compatible Intervals - PREM supports *Compatible Intervals* [18] for the execution of code that for some reason cannot be transformed into *Load*, *Execute*, and *Store* phases (e.g., syscalls). Code constructs that would require compatible intervals are uncommon on the GPU, but in a way, they represent the simplest way of ensuring memory isolation one could think of. The specification of a compatible interval is fulfilled after the insertion of the synchronization skeleton, as shown in Fig. 4, as the unmodified tile is executed in the Memory phase. Execution of the unmodified tile in the PREM Memory phase enables predictable scheduling of the Compatible Interval, but at the cost of complete idleness when the GPU does not have access to the DRAM. For the remainder of this section we refer to this scheme as CMPT.
Decoupled Access Execute - Decoupled Access Execute (DAE) [25] is a method to create PREM memory phases based on the *stripping* technique (as presented in Sec. 3.2.3), and in contrast to CMPT enables execution to continue even without access to main memory. The DAE technique represents the previous state of the art for the creation of PREM memory phases [16], and together with SDMA represents LES phase execution.

4.1 Evaluation methodology

The compiler transformations have been implemented in LLVM, using the OpenMP frontend as a hook to trigger the transformations. The NVPTX (NVIDIA Parallel Threads eXecution) backend is used to generate code for a concrete embodiment of the discussed architectural template, the NVIDIA Tegra TX1. The TX1 consists of a four-core ARM A57 CPU running Linux, and a two-cluster NVIDIA Maxwell GPU, with 128 physical cores each. Each GPU cluster has access to a $48kB$ local SPM, and all clusters share a L2 cache of $256kB$. The off-chip DRAM is shared with the CPU. It has been shown that the LLVM OpenMP v4 support [20], introduces less than 5% overhead compared to native CUDA when static loop scheduling is employed [29]. Our techniques extend this infrastructure, and our measurements confirm the published low-overhead numbers. As a representative GPU workload we consider kernels from the PolyBench-ACC suite [30], compiled with the *standard* data set for the best performing block/grid dimensions.

Our main goal is to achieve timing predictable execution, and we expect the required infrastructure and code transformations to introduce some overhead in the execution. Consequently, we divide our evaluation into two blocks. The first block, in Sec. 4.2, explores the performance impact of code transformation, synchronization, and changes in memory access patterns. Here we compare the novel SDMA both to CMPT and the state-of-the-art DAE versions, used in our previous work [16]. The second block, in Sec. 4.3, discusses predictability results for *SoftDMA* compared to the baseline OpenMP implementation without PREM.

All execution times we present have been normalized to that of the unmodified OpenMP baseline, and all reported timing results are the *measured* worst case execution times (WCET), as real-time systems must always be dimensioned to account for extreme cases. For all performance-related experiments we measure execution time in isolation (i.e., without memory interference), whereas for predictability results we measure execution time in presence of high memory interference. The memory interference from the CPU is generated using the *stress* [31] tool, which is able to produce large amounts of memory interference on a system.

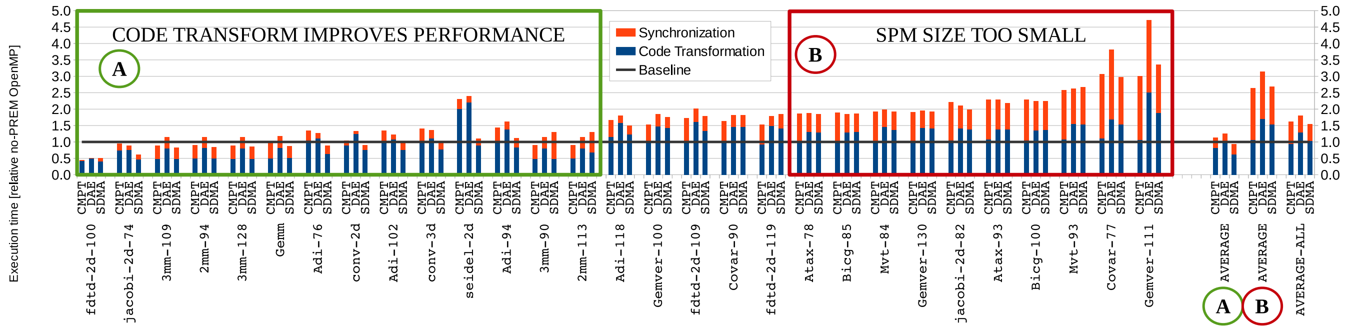


Fig. 7: The performance of code transformation and synchronization on the kernels, relative to the unmodified kernels.

4.2 PREM effects on performance

4.2.1 Overheads

Fig. 7 shows the effect of PREM code transformations and synchronization on the execution times of the kernels, compared to the baseline (unmodified kernel). Note that we are not co-scheduling the CPU and GPU at this point, only looking at the GPU performance in isolation. There are two main factors that influence the performance of the transformed kernels: The change in instruction count and memory accesses to support the PREM Memory phases, and the synchronization required to separate the PREM phases.

Instruction count and access patterns - Both DAE and SDMA add instructions to the CMPT scheme to implement the separation into *load*, *execute*, *store* phases, which is bound to introduce an overhead, as can be seen in the *blue part of the bars*. Note that, even in light of this, several benchmarks show a performance increase, which is discussed in detail at the end of this section. The DAE transformation is costlier than SDMA, as the minimal SDMA loops that are guaranteed to access each element only once require less complex control flow and fewer memory accesses than DAE. Instead, DAE reuses the original control flow of the kernel for the memory phase, which results in significant code duplication. This difference can clearly be seen in the benchmarks in the left side of Fig. 7. For several kernels in the right part of the figure, the slowdown in DAE and SDMA is similar, meaning either that the access pattern of DAE is similar to that of SDMA (e.g., no data reuse or non-coalesced access pattern that cause DAE overheads), or that the kernels have so little computation that even the smallest extension in instructions visibly impacts performance.

On average, the cost of the SDMA load and store phases is negligible, while the previously published DAE adds a 20% overhead. CMPT has no significant impact on instruction count, and impact on code performance is negligible. A special case is *seidel-2d*, where the original access pattern causes frequent stalls, amplified by tiling. SDMA regains this performance by reordering accesses to minimize stalls.

Synchronization - Together with the effects of code transformation, the *synchronization* overhead is also presented in Fig. 7, illustrated by *the red part of the bars*. The absolute synchronization cost S is equal for all kernels ($S_{measured} = 5.8\mu s$), and its impact on each kernel is determined by how well S is amortized with useful work T (see Sec. 3.3.1). T does not yet account for the WCET, covered in the next section. However, there is one further effect at play, which we refer to as the *synchronization wall*. This effect primarily affects the kernels in the right-most part of Fig. 7 (labeled B). As synchronization is initiated on the CPU at the expiry of timers, there is a maximum frequency at which the synchronizations can occur, based on the response time R_{timer} of the timer interrupt. Over 50000 measurements, $R_{timer} \leq 10.7\mu s$ for 95% of the cases, and $R_{timer} \leq 17\mu s$ for 99.9% of the cases[9] (including synchronization cost S).

For benchmarks where T of one or both of the PREM phases is below this value, $T < R_{timer}$, the GPU will idle for $R_{timer} - T$ time units at the synchronization point until the CPU responds (as with road traffic: the faster you arrive at the red light, the longer you have to wait). In Fig. 7 this manifests as the synchronization cost bringing the kernels up to a similar execution time. For some kernels, e.g., *covariance-77*, this effect is only visible for SDMA and CMPT – for DAE the instruction overhead dominates.

It has to be underlined that the *synchronization wall* effect does not highlight a limitation of the methodology *per se*. For these kernels, the SPM is simply not large enough to hold enough data for the phase lengths T to dominate the synchronization cost S (or, the maximum speed at which the CPU and the GPU can synchronize is too slow compared to the SPM refill rate). This problem intuitively disappears as local storage becomes larger[10], as shown in [21]. It is also worth noting, that because the sources of overhead are well understood, and their main influencing factors can be known at compile time [34], this can be used to inform the developer via compiler warnings when the performance degradation effects with HePREM will be noticeable.

On average, the performance impact of the synchronizations required to be able to execute predictably is about 50%, even when the kernels that hit the *synchronization wall* are included, and for some kernels it can be negligible. Overall, the synchronization cost is similar for all transformations.

Overall impact - No matter which transformation is used, it is evident across the board that the kernels can be divided into a group for which the transformations do not impact (or improve) performance, grouped to the left in Fig. 7 and labeled A, and those for which performance is degraded. The main factor that determines if performance increases is the amount of data reuse, i.e., the temporal locality of the computation. The data reuse factor is of importance to any transformation that tries to gain performance by better use of local memories, such as cache or SPM, as it

9. Outliers up to $R^{max}_{timer} = 97\mu s$ have been measured, but could be removed with the Linux *PREEMPT_RT* patches, and are not considered.

10. Platforms with larger SPM are available, e.g., Kalray [32]. We are currently exploring HERO [33], providing a 256 KB SPM for 8 cores, as opposed to 48 KB for up to 1024 CUDA threads. The latest NVIDIA GPUs have 96KB SPMs, but kernel blocks are still limited to 48KB.

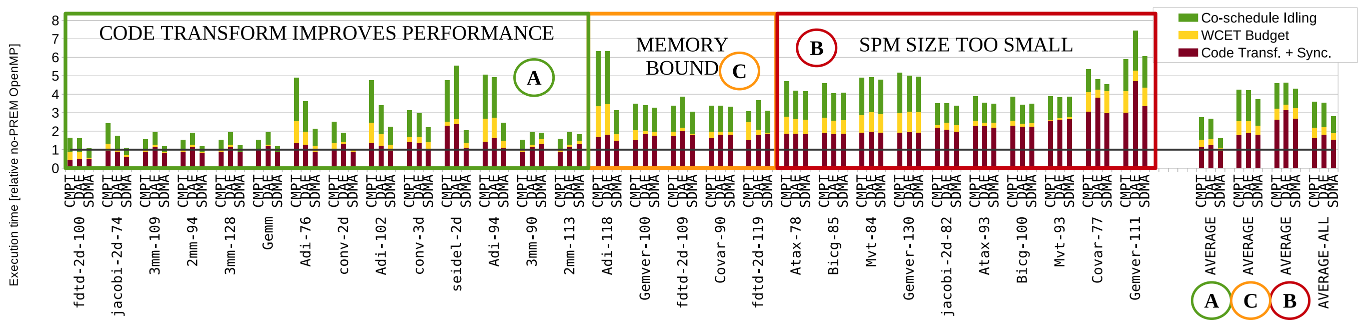


Fig. 8: The idling introduced into the program due to enforcing the *Fair* system memory schedule.

is only when the accessed data is already available locally that the caching benefits come into play. Thus, the kernels with a higher amount of data reuse will show benefits from tiling, which is the fundament for all the presented transformations. The performance illustrated by the blue bars for the CMPT transformation directly measures how effective the *tiling* transformation is for each benchmark, as it is the only transformation done in this case. We note that for some kernels, such as *conv-2d*, the optimized memory accesses of SDMA further improves the result.

We conclude that the novel SoftDMA always performs better than (or on-par) the DAE transformation used in previous work [16], [25], allowing on average 34% faster execution for kernels labeled A, and 17% over all kernels.

4.2.2 Idleness insertion due to System Co-scheduling

PREM requires enforcing the WCET for each phase before triggering the synchronization that precedes the beginning of a new phase[11]. Under these conditions, Fig. 8 shows the achieved execution times when scheduling exclusive memory between the CPU and the GPU. The relative execution times are broken down into three parts, where purple is the execution time of code and synchronization previously shown in Fig. 7. The remaining segments present two different types of idling introduced due to scheduling. The yellow segments show the idling introduced due to WCET budgeting, i.e., ensuring that the schedule has enough slack so that the PREM phase finishes also under the WCET. The green segments show the idle time in the system when sharing the memory bandwidth equally between the CPU and the GPU, which we will return to shortly. Lastly, in addition to the A and B categorizations introduced in Fig. 7, Fig. 8 introduces an additional category C of memory bound benchmarks, which will require special consideration.

Best case – Beginning with the purple plus yellow segments, we can read out the peak performance for each kernel that can be achieved while *guaranteeing* that it will never miss its deadline. This is achieved by reserving enough time in T to encompass the WCET, and by setting $E = T$ it contains the smallest amount of budgeting, and thus idling, possible. For this reason, we refer to this as the *best case* schedule.

This budgeting impacts the kernels differently, and for many kernels (*adi*, *atax*, *bicg*, *mvt*, *gemver*, and *covariance*) we see that the cost of enforcing the worst case phase length can cause a considerable slowdown. However, this effect can be significantly reduced when using SDMA, as the optimized memory phases streamline the memory accesses, giving raise to less variance in the execution time between invocations. This effect is large enough to slightly affect the average, and significantly contributes in for example the *adi* kernels. In other kernels the memory access patterns do not contain any significant segments of sequential accesses that SDMA can leverage (due to, e.g., row-by-row accesses), as is most pronounced in *atax-78* to *gemver-130*. In this case SDMA can not improve the WCET budgeting, and the resulting access pattern is similar as in both CMPT and DAE.

Fair sharing – While the *best case* schedule introduces the least amount of idling, it might not be possible to achieve this performance in a realistic system, as it must be scheduled to provide memory access to tasks on the CPU as well.

This brings us back to the original reason to transform the code into separate load, execute, and store (LES) phases: We want to minimize the time that the kernel requires memory access, so that it can make progress while the CPU is accessing memory. Our main goal is therefore to establish that PREM delivers on the promise to increase performance by continuing execution even when memory access is not granted. To evaluate the improvement in the transformed code, we compare their performance when memory access is only granted to the GPU 50% of the execution time. To achieve fair 50/50 memory scheduling between the CPU and the GPU, we enforce the length of the longest PREM phase to both phases, i.e., $E_{memory} = E_{compute} = max(T_{memory}, T_{compute})$. The results are shown with the full bars (purple, yellow, and green) in Fig. 8.

We note that DAE performs worse or at best on-par with the novel SDMA scheme, which is due to DAEs inefficiency in redistributing execution time from T_{memory} to $T_{compute}$. This is because the *stripping* technique creates un-optimized access patterns that might even fetch data multiple times, as outlined in Sec. 3.2.3. Because $T^{SDMA}_{compute} = T^{DAE}_{compute}$ (same transformation), and $T^{SDMA}_{memory} \leq T^{DAE}_{memory}$, SDMA improves over the state-of-the-art also under co-scheduling with the CPU, reducing idle time by 45% under *fair sharing*.

Having established that SDMA improves over the other LES scheme, we compare SDMA with CMPT. Since CMPT only executes in the *Memory* phase, $T^{CMPT}_{compute} = 0$, it will always idle for half of the time. Fig. 8 shows that on average SDMA introduces about half as much idling (green segment) as CMPT, and even less in the kernels highlighted in the left of Fig. 8 (labeled A). These kernels have a good balance between memory and compute time and can therefore gain most from a balanced schedule. In the *convolution-*

11. The WCET of the phases are determined by recording failed synchronizations during kernel execution. This occurs when the timer interrupt was triggered on the CPU, but the GPU had not yet reached the synchronization point, and thus the synchronization could not be performed. We incrementally increase the timer timeout until no synchronization fails, at which point the delay accounts for the WCET.

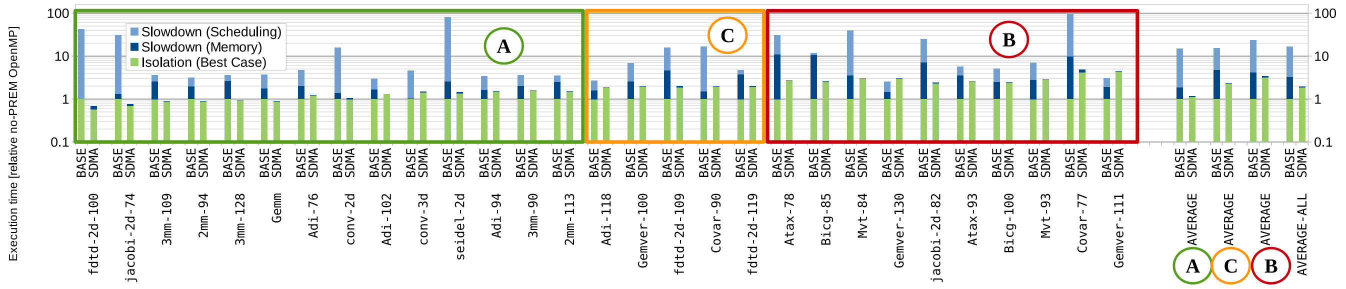


Fig. 9: The performance degradation due to memory interference from the CPU for the *best case* system schedule.

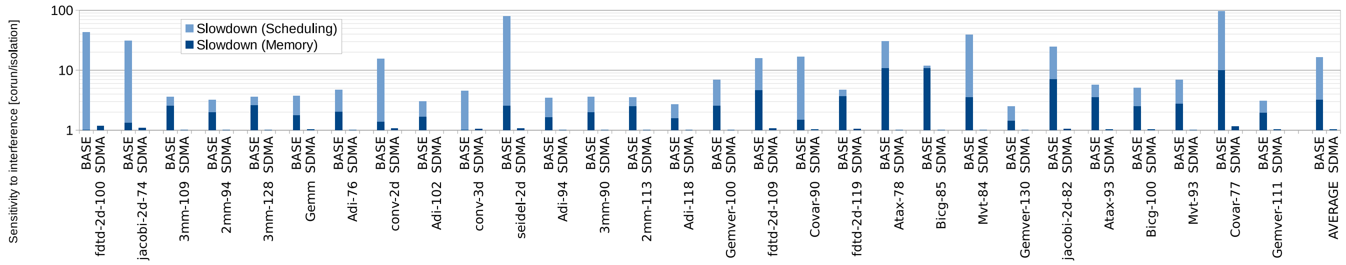


Fig. 10: The sensitivity to interference for the BASE and SDMA versions of the kernels.

2d kernel, co-schedule idling is near-zero (1.2%). For this kernel, $T_{memory} \approx T_{compute}$, which maps well to a fair sharing scheme with the CPU: Eq. 2 gives $E \approx T$, which introduces a low amount of idling I as given by Eq. 3. For most other kernels in the A set, we see similar results.

In contrast, kernels that hit the *synchronization wall* show a similar amount of idling for all transformations, as the execution time of the phases T is dominated by synchronization S, due to the small SPM memory. Between these, there is a set of kernels in the center of Fig. 8, labeled C, which show only marginal benefits from SDMA. These kernels are so memory bound that essentially no work is done in the Compute phase ($T_{memory} \gg T_{compute}$), in practice making the SDMA kernel execute in the same manner as *CMPT*.

Overall SDMA on average reduces the idling by 45% compared to DAE, and 53% compared to CMPT.

4.2.3 Performance summary

The key performance take-aways are: i) SDMA on average performs 20% better, and up to 48% better than the state-of-the-art DAE transformation. ii) For kernels where the PREM transformations are beneficial the SDMA average slowdown under fair sharing *is only* 59%*, and can be as low as* 1.2%*, significantly improving over* $2.74\times$ *slowdown in CMPT, and* $2.67\times$ *in DAE. iii) In heavily memory bound kernels, the inability to perform work in the compute phase increases the average slowdown to* $3.8\times$ *under* fair sharing*. In this case, PREM memory scheduling is unable to provide large improvements. iv) In kernels with phases shorter than the synchronization granularity, slowdown is on average* $4.3\times$*. This happens when the SPM is too small, and is an effect of the hardware used and not the technique itself.*

Given its superior performance and inferior idling, we only consider SDMA for the predictability experiments.

4.3 PREM effects on Predictability

To validate the effectiveness of the proposed SDMA toward guaranteeing *robustness to interference*, we execute the GPU kernels under heavy interference from the CPU, as outlined in Sec. 4.1. The results for the *best case* scenario are shown in Fig. 9. An additional BASE configuration has been added for each kernel, representing the execution time of the unmodified OpenMP program in presence of interference. We know from Sec. 3.3.2 that the interference consists of two parts, *memory interference* and *scheduling jitter*. While SDMA is not affected by the latter (see Sec. 3.3.2), to provide a fair comparison for BASE, we execute those kernels twice: We measure the *memory interference* by executing the offloading process once with the highest priority (i.e., *low niceness*), and measure CPU *scheduling jitter* by executing it at the same priority as the interfering process. The green part of the bars shows execution time in isolation, while the blue and cyan parts shows additional execution time due to interference, from memory and Linux scheduling respectively.

Results show that the performance of BASE is degraded to such a high degree that the SDMA kernels perform better in almost all cases, despite the factors of slowdown presented previously. On average, SDMA results under interference remain similar as in isolation ($+3.5\%$), which is 7 times better than BASE under both interference types, and 67% better when only considering memory interference.

As *robustness to interference* is the most important feature of PREM, it is further highlighted in Fig. 10, which shows execution time under interference for BASE and SDMA normalized to the execution time in isolation for the same scheme. The low variance in execution time of SDMA, on average 3.5%, is greatly contrasted to BASE where performance can degrade by orders of magnitude. The interference to the baseline is based on measurements, and as such provides a lower bound on the interference that can be experienced. In contrast, the Predictable Execution Model [12] provides *robustness to interference* by design, which means that near-zero interference is indeed the expected value.

5 Related Work

This work extends the Predictable Execution Model (PREM), the key novelty in our proposal being the extension to GPUs and arbitration of the accesses to shared resources in a heterogeneous embedded system. PREM was originally proposed in for single-core CPUs [18] to provide *robustness to interference* from peripheral devices, and was later extended to counter inter-core interference in multi-core CPUs [12].

Previous work on heterogeneous architectures has mainly focused on *discrete* GPUs[12], and only very recently on *integrated* GPUs. For discrete CPU-GPU systems the point of interference is reduced to data movements between the discrete CPU and GPU memories over the *PCIe* bus. In this context, several approaches have been proposed [35] [36] [37]. Concerning *integrated* GPUs, the point of interference is not limited to a single peripheral bus, as the memory hierarchy itself is largely shared between the two devices.

To achieve timing predictability in such systems, BWLOCK++ [38] extends the *throttle thread* based priority mechanism in BWLOCK [39] to include memory throttling in the context of GPUs, triggered at offloading time. BWLOCK++ limits the interference to GPU programs by throttling CPU tasks after a threshold value of memory accesses has been exceeded. Like HePREM, it targets the reduction of memory interference on GPU kernels, but it operates on unmodified legacy code. In contrast to PREM-based solutions like ours, BWLOCK++ cannot provide system-level *robustness to interference*, as CPU tasks are seen as lower-criticality tasks that can be arbitrarily throttled.

Throttle threads as a concept were first proposed in *MemGuard* [27], which uses a memory bandwidth budgeting scheme, triggering hardware performance counters (PMC) interrupts to schedule *throttle threads* once the budget is exhausted. Relying on the PMC, this technique is only suitable for CPUs. This work reuses *throttle threads* by significantly extending our previous work on *GPUguard* [40] (Sec. 3.3) from a proof of concept to a formalized component in PREM CPU-GPU scheduling. Similarly to *GPUguard*, *SiGAMMA* [41] employs high-priority CUDA *spin kernels* (i.e., a “GPU throttle thread”) on the GPU to ensure that the GPU does not interfere with the CPU. Compared to what we propose, *SiGAMMA* looks at the dual problem of reducing interference to CPU real-time tasks, the GPU being the adversary.

Previous work on PREM compiler support includes the work by Soliman et al., who propose a compiler analysis to detect program regions suitable for PREM intervals for SPM-equipped CPUs [42]. This analysis only handles static allocation of entire data structures (e.g., arrays), which limits its applicability to small datasets that fit in the L1 SPM. In contrast, our approach applies *tiling* to partition large data structures, significantly extending its applicability. Our work focuses on GPUs, rather than CPUs, and to the best of our knowledge, our work is the first on PREM for GPUs.

Separating memory and computation has also been studied outside the context of PREM, to provide performance or energy benefits to CPUs. Relevant examples are *Decoupled-access-execute* [25] and *Clairvoyance* [43], which were proposed as means of i) applying dynamic voltage-frequency scaling (DVFS) independently for memory and compute tasks; and ii) increasing the memory and instruction level parallelism for limited out-of-order cores. This method was used to implement code *stripping*, which was at the heart of our previous work [16], but as shown in Sec. 4 it is not suitable for GPUs, leading us to propose *SoftDMA*.

SoftDMA improves GPU performance by decoupling threads performing memory accesses from thread(s) that use the data for computation. A conceptually similar decoupling is used in *warp specialization* [26]. In comparison, *SoftDMA* provides two additional benefits: i) it decouples memory from computation by separation in time rather than in space (specialization of thread workloads): matching standard work distribution primitives (e.g., OpenMP *teams* and *threads*); ii) the creation of SoftDMA memory phases can be created automatically based on compile time information.

The division of loops into fine-grained schedulable intervals in HePREM is based on *tiling*, for which there are several types of analysis that modern compilers can leverage. This work uses the built-in LLVM *scalar evolution analysis* [44]. Newer and more powerful tools such as polyhedral analysis [23] [45] are reaching full maturity and are being included in modern compilers. Our methodology is not tied to a particular tiling technique, and work on optimized tiling techniques are complementary to this work.

12. GPUs with their own memory system, not shared with the CPU.

6 Conclusion

We have presented HePREM, which extends the Predictable Execution Model (PREM) to shared memory heterogeneous SoCs. A timer-based synchronization mechanism enforces mutual CPU/GPU exclusion to shared DRAM, thus enabling timing predictable execution on COTS hardware. To support PREM execution of GPU programs, we propose *SoftDMA*, a compiler transformation capable of creating optimized GPU memory phases from high-level languages. *SoftDMA* uses loop analysis, tiling, and access redistribution and reordering over threads to improve coalescing.

HePREM is able to limit CPU interference to GPU programs to near-zero values, even under heavy memory use. The code transformations and system scheduling to support this on average imply an overhead of $2.8\times$ when sharing the memory bandwidth 50/50 between the CPU and the GPU. However, for compute bound kernels, overheads are measured as low as 1.2%. The high average arises from costly synchronizations in memory bound kernels, due to small local memories on the NVIDIA TX1 evaluation system.

Having shown in this work that PREM on the GPU is feasible, we are as part of our ongoing work exploring the co-scheduling of fully PREM-compliant tasks on both CPU and GPU. For this, we are also looking into automatic generation of PREM code from generic CPU code to automate the process. For kernels that do not perform well under PREM on the NVIDIA TX1, we are exploring platforms with larger SPMs that are able to amortize the increased synchronization cost in heterogeneous PREM systems.

Acknowledgment

This work has been supported by the EU H2020 project HERCULES (688860).

References

[1] P. Burgio *et al.*, “A software stack for next-generation automotive systems on many-core heterogeneous platforms,” in *Digital System Design (DSD)*. IEEE, 2016.
[2] W. Shi *et al.*, “Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey,” *Integration, the VLSI Journal*, vol. 59, pp. 148 – 156, 2017.
[3] A. Skende, “Introducing Parker: Next-generation Tegra System-on-Chip,” in *2016 IEEE Hot Chips 28 Symposium (HCS)*, Aug 2016.
[4] “AMD Embedded G-Series LX,” Mar 2018. [Online]. Available: https://www.amd.com/en/products/embedded-g-series-lx

[5] S. Mittal *et al.*, "A survey of methods for analyzing and improving GPU energy efficiency," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 19:1–19:23, Aug. 2014.

[6] M. Dehyadegari *et al.*, "Architecture support for tightly-coupled multi-core clusters with shared-memory HW accelerators," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2132–2144, Aug 2015.

[7] R. Cavicchioli *et al.*, "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms," in *ETFA'17*, Sept 2017, pp. 1–10.

[8] F. Zhang *et al.*, "Understanding co-running behaviors on integrated CPU/GPU architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 905–918, March 2017.

[9] M. D. Gomony *et al.*, "A globally arbitrated memory tree for mixed-time-criticality systems," *IEEE Trans. on Computers*, vol. 66, no. 2, Feb 2017.

[10] D. Dasari *et al.*, "A framework for memory contention analysis in multi-core platforms," *Real-Time Systems*, vol. 52, no. 3, May 2016.

[11] I. Agirre *et al.*, "On the tailoring of CAST-32A certification guidance to real COTS multicore architectures," in *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2017.

[12] A. Alhammad *et al.*, "Time-predictable execution of multithreaded applications on multicore systems," in *DATE'14*. IEEE, 2014.

[13] OpenMP Architecture Review Board, "OpenMP application programming interface version 4.5," Nov 2015. [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[14] "The OpenACC application programming interface version 2.6." [Online]. Available: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf

[15] J. Bosch *et al.*, "Exploiting parallelism on GPUs and FPGAs with OmpSs," in *1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC Systems*. ACM, 2017.

[16] B. Forsberg *et al.*, "HePREM: Enabling predictable GPU execution on heterogeneous SoC," in *DATE'18*, 2018.

[17] "NVIDIA Jetson." [Online]. Available: http://www.nvidia.com/object/jetson-tx1-dev-kit.html

[18] R. Pellizzoni *et al.*, "A predictable execution model for COTS-based embedded systems," in *RTAS'11*. IEEE, 2011.

[19] Motor Industry Software Reliability Association and Motor Industry Software Reliability Association Staff, *MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems*. Motor Industry Research Association, 2013.

[20] S. F. Antao *et al.*, "Offloading support for OpenMP in Clang and LLVM," in *LLVM-HPC'16*, 2016.

[21] B. Forsberg *et al.*, "Taming data caches for predictable execution on GPU-based SoCs," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 650–653.

[22] R. Allen *et al.*, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[23] T. Grosser *et al.*, "Polly performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, 2012.

[24] S. Sioutas *et al.*, "Loop transformations leveraging hardware prefetching," in *CGO'18*, February 2018.

[25] K. Koukos *et al.*, "Multiversioned decoupled access-execute: The key to energy-efficient compilation of general-purpose programs," in *25th Int. Conf. on Compiler Construction*. ACM, 2016.

[26] M. Bauer *et al.*, "CudaDMA: optimizing GPU memory bandwidth via warp specialization," in *High performance computing, networking, storage and analysis*. ACM, 2011.

[27] H. Yun *et al.*, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Real-Time and Embedded Techn. and Appl. Symp. (RTAS)*. IEEE, 2013.

[28] Q. Zhu *et al.*, "Understanding co-run degradations on integrated heterogeneous processors," in *Languages and Compilers for Parallel Computing*, 2015.

[29] G.-T. Bercea *et al.*, "Performance analysis of OpenMP on a GPU using a CORAL proxy application," in *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. ACM, 2015.

[30] S. Grauer-Gray *et al.*, "Auto-tuning a high-level language targeted to GPU codes," in *2012 Innovative Parallel Computing (InPar)*, 2012.

[31] FreshPorts – sysutils/stress: Tool to impose load on and stress test Unix-like systems. [Online]. Available: https://www.freshports.org/sysutils/stress/

[32] B. D. de Dinechin, "Kalray MPPA: Massively parallel processor array: Revisiting DSP acceleration with the Kalray MPPA manycore processor," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Aug 2015.

[33] A. Kurth *et al.*, "HERO: Heterogeneous embedded research platform for exploring RISC-V manycore accelerators on FPGA," *arXiv*, 2017.

[34] B. Forsberg *et al.*, "On the cost of freedom from interference in heterogeneous SoCs," in *21st International Workshop on Software and Compilers for Embedded Systems*. ACM, 2018.

[35] G. A. Elliott *et al.*, "GPUSync: A framework for real-time GPU management," in *2013 IEEE 34th Real-Time Systems Symp.*, 2013.

[36] Q. Chen *et al.*, "Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," in *ASPLOS'16*. ACM, 2016.

[37] Y. Suzuki *et al.*, "Real-time GPU resource management with loadable kernel modules," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1715–1727, June 2017.

[38] W. Ali *et al.*, "Protecting Real-Time GPU Applications on Integrated CPU-GPU SoC Platforms," *ArXiv e-prints*, Dec. 2017.

[39] H. Yun *et al.*, "Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms," *IEEE Trans. on Computers*, vol. 66, no. 7, 2017.

[40] B. Forsberg *et al.*, "GPUguard: Towards supporting a predictable execution model for heterogeneous SoC," in *DATE'17*, 2017.

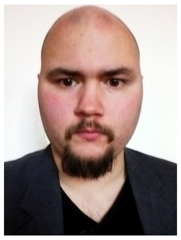
[41] N. Capodieci *et al.*, "SiGAMMA: Server based integrated GPU arbitration mechanism for memory accesses," in *25th International Conference on Real-Time Networks and Systems*. ACM, 2017.

[42] M. R. Soliman *et al.*, "WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching," in *ECRTS'17*, 2017.

[43] K.-A. Tran *et al.*, "Clairvoyance: Look-ahead compile-time scheduling," in *CGO'17*. IEEE, 2017, pp. 171–184.

[44] R. A. van Engelen, "Efficient symbolic analysis for optimizing compilers," in *Compiler Construction*, R. Wilhelm, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 118–132.

[45] K. Trifunovic *et al.*, "GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation," in *GROW'10*, Pisa, Italy, Jan. 2010.



Björn Forsberg received his M.Sc. degree in Information Technology Engineering from Uppsala Universitet, Sweden, in 2015. Since then he has been at the Swiss Federal Institute of Technology Zürich, Switzerland where he is currently pursuing his Ph.D. degree. His current interests include timing predictable real-time execution on commercial-of-the-shelf hardware, with special focus on emerging heterogeneous SoCs, and the programming models and compilers required to support such systems.

Luca Benini holds the chair of Digital Circuits and Systems at ETHZ and is Full Professor at the Universita di Bologna. He received his PhD degree from Stanford University in 1997. Dr. Benini's research interests are in energy-efficient system design, from embedded to HPC. He is also active in the design of smart sensing micro-systems and ultra-low power VLSI circuits. He has published more than 900 peer-reviewed papers and five books. He is a Fellow of the ACM and a member of the Academia Europaea. He received the 2016 IEEE CAS Mac Van Valkenburg award.

Andrea Marongiu received the PhD degree in electronic engineering from the University of Bologna, Italy, in 2010. He has been a postdoctoral reserch fellow at ETH Zurich, Switzerland. He currently is an associate professor at the University of Modena and Reggio Emilia. His research interests focus on programming models and architectures in the domain of heterogeneous multi- and many-core systems-on-chip. In this field, he has published more than 100 papers in peer-reviewed conferences and journals.