HePREM: Enabling Predictable GPU Execution on Heterogeneous SoC

Björn Forsberg[1] Luca Benini[1, 2] Andrea Marongiu[1, 2]
[1] Swiss Federal Institute of Technology Zürich [2] University of Bologna
{bjoernf, lbenini, a.marongiu}@iis.ee.ethz.ch

***Abstract*—Heterogeneous systems-on-a-chip are increasingly embracing shared memory designs, in which a single DRAM is used for both the main CPU and an integrated GPU. This architectural paradigm reduces the overheads associated with data movements and simplifies programmability. However, the deployment of real-time workloads on such architectures is troublesome, as memory contention significantly increases execution time of tasks and the pessimism in worst-case execution time (WCET) estimates. The Predictable Execution Model (PREM) separates memory and computation phases in real-time codes, then arbitrates memory phases from different tasks such that only one core at a time can access the DRAM. This paper revisits the original PREM proposal in the context of heterogeneous SoCs, proposing a compiler-based approach to make GPU codes PREM-compliant. Starting from high-level specifications of computation offloading, suitable program regions are selected and separated into memory and compute phases. Our experimental results show that the proposed technique is able to reduce the sensitivity of GPU kernels to memory interference to near zero, and achieves up to a $20\times$ reduction in the measured WCET.**

I. Introduction and Related Work

Heterogeneous systems-on-a-chip (SoC) are been increasingly adopted to satisfy the high performance and energy efficiency demands of sophisticated applications, e.g., from the automotive and avionics domains. The most common architectural template consists of a general-purpose *host* processor and an integrated GPU (iGPU), physically sharing a single main memory (usually DRAM). This reduces the overhead of data transfers and greatly simplifies the programming model, at the cost of making the execution time of the deployed workloads sensitive to memory contention and interference. When executing several benchmarks from the PolyBench-ACC suite[1] [1] on an NVIDIA Tegra TX1 under memory interference, we observed order-of-magnitude slowdowns for GPU kernel execution. Such a heavy perturbation makes the adoption of commercial off-the-shelf (COTS) devices difficult for the construction of real-time systems, where correct timing behavior must be ensured under all conditions. Given the complexity of such designs, deriving worst-case execution time (WCET) estimates based on analytical approaches requires overly pessimistic assumptions to account for access time variability, causing poor utilization.

As architectural support to mitigate the effect of memory contention [2] [3] is typically not available in COTS heterogeneous platforms, the real-time community is actively exploring several software techniques to enable predictable execution (i.e., freedom from interference in CPU-GPU co-runs). Focusing on *host* CPU management, memory arbitration mechanisms have been recently proposed to co-schedule memory and processing bandwidth by multiple cores, such as MEMGUARD [4], BWLOCK [5] and the predictable execution model (PREM) [6] [7]. Software techniques for heterogeneous SoC management have started appearing only very recently, and include scheduling of DMA memory transfers and kernel executions independently at offload time [8], [9] and preliminary explorations related to the suitability of the PREM in this context [10] [11].

PREM lowers the pessimism in WCET estimates by first separating memory and compute phases in real-time applications, and then scheduling the memory phases such that only one core at a time accesses the shared resource. The key difficulties in implementing this model are thus related to i) providing compiler techniques to restructure applications written with standard programming models; ii) designing centralized memory arbitration methods.

Concerning the first point, guidelines for compiler support for PREM were discussed in the original publication [6] (assuming a programmer-aided approach relying on directives), and a fully automatic approach has been proposed for scratchpad-based, general-purpose CPUs [12]. Techniques such as *decoupled access-execute* (DAE) [13] and *Clairvoyance* [14] also target the separation of memory and compute phases, but not in the context of real-time systems, and as such do not provide enforcement of the separation of the phases, nor require the data to reside locally to the processor. No work has been done so far on PREM compilation for GPUs.

Concerning the second point, the previously cited works that target real-time execution provide mechanisms to arbitrate memory requests between multiple CPUs, typically implemented as an extension to the operating system. In the context of heterogeneous SoCs, *GPUguard* [11] was recently proposed as a proof-of-concept synchronization scheme to support PREM for iGPUs, further described in Sec. III-A.

In this paper, we present HePREM, which revisits and extends the original concepts for predictable execution on heterogeneous SoCs. With HePREM we make thefollowing contributions: 1) we present a compiler-based technique for PREM compilation for iGPU kernels on heterogeneous SoCs; 2) we describe the first complete implementation of an integrated framework for PREM execution on heterogeneous SoCs; 3) we discuss an extensive set of experimental results aimed at showing the benefits of PREM in this context as well as at identifying the key bottlenecks. In particular the experiments demonstrate that the proposed approach limits the effect of memory interference on the WCET to near-zero, offering on average an order of magnitude reduction in performance loss compared to the unmodified GPU program.

II. HePREM Design

PREM assumes a program execution model with three main features: (i) applications are divided into a sequence of non-preemptive scheduling intervals; (ii) these scheduling intervals

[1] More information about the experimental setup is provided in Section IV.

978-3-9819263-0-9/DATE18/©2018 EDAA

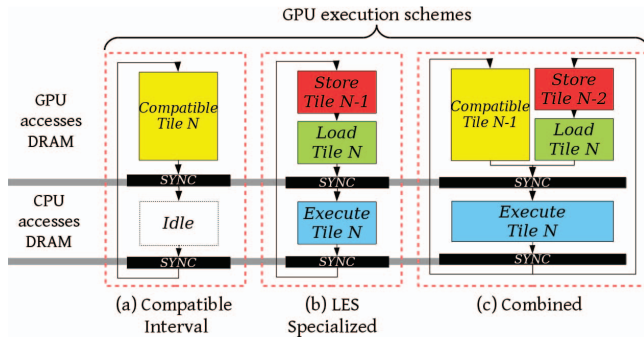


Fig. 1: A logical view of different GPU execution schemes enabling mutually-exclusive access to the system DRAM with the CPU executing in background.

(named *predictable intervals*) are executed predictably without OS calls and cache-misses by prefetching all required data at the beginning of the interval itself; (iii) the execution time of *predictable intervals* is kept constant by monitoring CPU time counters at run-time.

To implement *predictable intervals*, application code is re-arranged into memory phases, during which all the data required for the computation is moved to a local memory, and compute phases, which operate on local data. The model also considers *compatible intervals*, which are parts of the program that are compiled and executed without any special provisions (i.e., they are backwards compatible with legacy code). In these intervals, all required OS calls can be grouped and cache misses can happen at any time, provided that bounds on execution time can be computed based on static analysis techniques. Ideally, there should be a small number of *compatible intervals* which are kept as short as possible.

To provide freedom from interference, PREM enforces a coscheduling mechanism that serializes memory access requests from various actors (e.g., memory phases or *compatible intervals* from programs/tasks running on different cores). Since PREM only allows a single task to have its memory requests serviced at a time, other tasks might have to wait for their memory phase to begin, introducing *idling*. This also happens when a task finishes computation on its local data earlier than budgeted for the associated *predictable interval* (see point (iii) above).

A. HePREM Overview

HePREM extends the ideas of the original PREM proposal to GPU execution and compilation. In this context, a revisited notion of *compatible intervals* is the simplest form of GPU code transformation for predictable execution (freedom from interference). HePREM *compatible intervals* consist of parts of offloaded GPU kernels, at the boundaries of which synchronization with a central system arbiter is inserted. Since GPU code is loop-centric, techniques such as *loop tiling* [15] can be applied to effectively control granularity of scheduling intervals. Similar to the original idea of *compatible intervals*, the code within scheduling intervals (i.e., the content of a tiling loop) is unmodified, and cache misses can happen at any time. While HePREM *compatible intervals* imply minimal modifications to kernel code, which in turn implies minimal overhead, they are subject to performance drop due to high idleness, as the task comes to a complete stall while the program does not have access to main memory. This is shown in Figure 1a.

HePREM implements *predictable intervals* by first applying loop tiling to tune the granularity of scheduling intervals and then separating the code within scheduling intervals into specialized *load*, *execute* and *store* (LES) code. While intuitively LES-specialized *predictable intervals* imply higher code transformation overheads, they reduce the idleness in the system, as shown in Figure 1b.

HePREM also allows to specialize GPU kernel code to implement a *combined* execution, where half of the threads executes one *compatible interval* tile (directly accessing main memory via cache misses) and the other half executes a LES-specialized tile (moving data from main memory to local storage in the *memory phase*, then computing locally in the *compute phase*). This is shown in Figure 1c.

In the following, we discuss i) where, in the internal memory hierarchy, HePREM implements the buffering of the data for LES-specialized *predictable intervals*; ii) at which granularity, at the system level, HePREM implements synchronization points between the iGPU and the CPU; iii) how HePREM compilation is seamlessly integrated within standard programming models.

1) Staging through the scratchpad: Most existing heterogeneous SoCs offer two options for local storage: the scratchpad memory and the local cache hierarchy. The use of caches simplifies the addressing scheme, as the placement of data is handled in hardware, which requires less restructuring of the code for PREM compliance. In addition, the local L2 cache is generally larger than the size of the scratchpad memory. On the other hand, caches are inherently less predictable, as they are subject to self-eviction, which could cause cache misses during the PREM compute phase. This breaks the isolation property of PREM and would require additional mechanisms to prevent/control undesired evictions[2]. For these reasons HePREM targets the scratchpad memory as local storage.

2) Host-to-GPU synchronization: To realize PREM on a heterogeneous SoC, the *memory phases* need to be co-scheduled between the CPU and the GPU, which implies synchronization. To this end, it might be tempting to leverage the existing infrastructure for GPU offloading. At a low level, the offloading mechanism is simply the submission of jobs to a buffer, from which the GPU issues these to the hardware. It is therefore possible to control the entering of GPU *memory* and *compute phases* by instrumenting the submission of the jobs to the buffer. However, this has significant drawbacks. First, changes would be required to the GPU drivers to discipline the execution of *memory* and *compute phases* on the GPU, according to the global scheduler's decision. GPU drivers are generally not available as open-source components, and those that are generally suffer from poor performance and low reliability due to their experimental stage of development. Second, the *memory* and *compute phases* of embedded real-time tasks might be extremely small, which makes it questionable if the execution of the offloading mechanism at such fine granularity is even feasible due to the large overheads it would introduce. Third, as the loading of data to the local storage can only be done from the GPU itself, the memory/compute separation cannot be done at kernel granularity, as local memories are cleared at kernel boundaries.

[2]It is worth noting that the required harware to implement, e.g., cache locking, is often not available on such platforms.

For this reason, HePREM encodes the phase switches directly in the GPU code. This is subject to much smaller overheads, as only a single interaction via the GPU driver is required, and provides much higher degree of control.

3) High-level language compilation: In recent years, programming models have evolved to support offloading computations to data-parallel accelerators such as GPUs. Those that were explicitly created for modern GPUs (e.g., CUDA and OpenCL) are often too low-level and require significant programmer involvement to restructure the data-parallel computation in a way that maps efficiently to the GPU hardware. Other approaches (e.g., OpenMP and OpenACC) try to abstract this process by relying on higher-level constructs – typically in the form of compiler directives – to offload the execution entire loop nests to the GPU.

Besides the increased ease of programming, this approach has an additional benefit when it comes to implementing PREM. As the OpenMP compiler itself is tasked with workload distribution, data movements, and the generation of parallel code from high level loop description, the visibility of the complete iteration space gives the compiler full freedom in distributing and reorganizing computations to identify suitable PREM regions. The same is not possible, e.g., in OpenCL, without having to undo decisions made by the programmer in terms of iteration space partitioning and distribution.

Because of this, HePREM compilation is implemented as part of high-level loop offloading mechanisms.

III. HePREM Implementation

HePREM is based on a i) synchronization middleware and ii) compiler support, which we describe in the following subsections.

A. Synchronization middleware

The CPU/GPU synchronization mechanism of HePREM is inspired to GPUguard [11]. Since COTS heterogeneous SoCs lack mechanisms to trigger CPU-side software interrupts from GPU code, the enforcement of the length of the system-level memory and compute phases are managed from the CPU, which has access to hardware timer interrupts. These are used for triggering the CPU-side synchronization handler, implemented in a Linux Loadable Kernel Module (LKM).

For the communication between the devices, messages are passed through a per-*cluster*[3] *synchronization channel* in the DRAM, which is visible from both the CPU and the GPU. The synchronization timer handler will ensure that the channels of all active clusters contains the relevant synchronization flag, and at that point it will correcly configure the CPU system for the next phase, whereafter it writes an ACKNOWLEDGEMENT flag to each active channel. For entering the PREM memory phase, the ENTERMEM synchronization command is encoded at the beginning of the GPU memory phase, and likewise, the ENTERCOMP at the beginning of the GPU compute phase. The GPU execution is then stalled through polling until the CPU has acknowledged the synchronization, signifying that the memory access token has been passed in accordance with the synchronization request. As the GPU cannot continue into the next phase until the synchronization request has been acknowledged by the CPU, this provides system-wide enforcement of the PREM isolation property.

As the focus of this work is on the transformation of GPU programs to conform to the PREM standard, using high-level programming models, the transformation of CPU programs is out of the scope of this work. However, to ensure that the memory isolation property of PREM holds, it has to be ensured that the CPU does not initiate memory accesses during the GPU memory phase. In place of PREM-compatible CPU programs, the throttle thread mechanism of MemGuard [4] is employed, which schedules a high-priority POSIX real-time thread that pre-empts the currently executing threads. The throttle thread performs busy waiting, and thus ensures that no further memory requests are generated from the core, and can thus be scheduled during the CPU compute phase to ensure that no global memory requests are generated.

B. Compiler Support

HePREM compilation is divided into three major steps: *Analysis*, *Refactoring* and *Transformation*, as shown in Figure 3. The steps are further described in this section.

1) Footprint Analysis and Scheduling Interval Selection: The main structure that the compiler operates on is loop nests. To identify scheduling intervals suitable for LES specialization, we rely on *scalar evolution* analysis to extract the range of data elements accessed in loops. From there we calculate the memory footprint for the current loop nest, knowing that i) the computation contained within outer nesting levels is loop invariant w.r.t. the footprint analysis; ii) it is guaranteed that inner loop nests have smaller or equal footprint. Based on these properties, we implement the algorithm described in Figure 2.

The recursive algorithm determines the memory footprint of the current loop nest level. If the entire loop fits into the scratchpad, then the full loop is simply selected as a single scheduling interval. In the same manner, if exactly one iteration fits in the scratchpad, the single iteration is selected as a scheduling interval. If multiple iterations (but not the entire loop) fit, loop tiling is applied to outline the exact number of iterations that fit into the scratchpad, as shown in Figure 3b. The tiled loop is selected as a scheduling interval. If a single

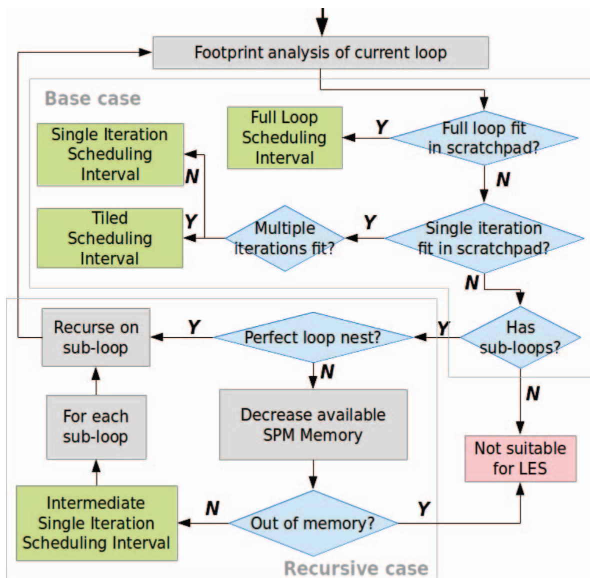


Fig. 2: Hierarchical memory footprint and scheduling interval selection algorithm.

[3] A *cluster* is equivalent to a *compute unit* in the OpenCL terminology, or a *streaming multiprocessor* in the CUDA terminology.

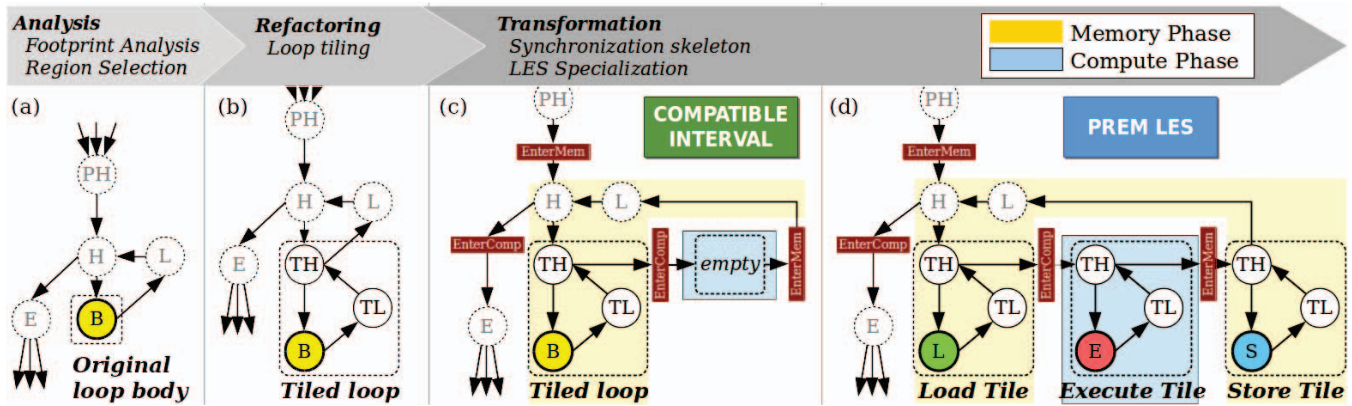


Fig. 3: The original loop (a), followed by the transformations performed during the compilation process: The loop is tiled (b), followed by the insertion of synchronization points (c), and the specialization into Load, Execute, and Store phases (d). For the transformation steps, the memory and compute phases are highlighted. Legend: B = Loop Body, PH = Pre-header, H = Header, L = Latch, E = Exit, TH = Tile Header, TL = Tile Latch, L/E/S = Load/Execute/Store specialized version of loop body.

iteration does not fit and there are sub-loops, the algorithm is called recursively on the sub-loops. Before the recursive call the available scratchpad space is decreased in accordance with the memory requirements of the current level (not considering sub-loops), which is selected as an intermediate scheduling interval. Thus, this interval becomes the parent of those of the recursed upon loops. If there are multiple sub-loops they are considered in isolation, as the memory use of sub-loops only depends on their parents. If the recursion continues to the level that the memory use is higher than the size of the scratchpad, or if a single iteration in the innermost loop does not fit, the loop cannot be staged through the scratchpad through tiling[4].

2) Synchronization Stub Insertion: Once scheduling intervals have been outlined, the synchronization stubs (ENTERMEM and ENTERCOMP) to implement the protocol described in Section III-A are inserted into the code, as shown in as shown in Figure 3c.

According to PREM, every program region should first load data from the global memory into local storage, corresponding to a PREM *memory phase*, and thus, we insert a ENTERMEM synchronization upon entry into the scheduling interval. Scheduling intervals will furthermore always be entered from the compute phase of the parent interval, and to keep consistent state a ENTERCOMP synchronization is inserted upon exit. To encode the *compute phase*, another set of ENTERCOMP and ENTERMEM synchronizations are inserted within the loop body. With no further changes, this implements a *compatible interval*, as shown in Figure 1a, where the original computation from the outlined scheduling interval is executed in the *memory phase*, and the *compute phase* is completely empty.

3) LES Specialization: The final transformation step clones the outlined scheduling interval code into three copies, each of which is specialized to perform *load*, *execute*, and *store*, as shown in Figure 3d. Here, *load* and *store* constitute the PREM *memory* phase, and *execute* maps directly to the PREM *compute* phase. For the *load* phase, all instructions that are not related to the loading of data are removed, so that only the loads themselves and instructions for address calculations are preserved. Then, all loaded data is stored into the scratchpad, using the same index calculation, modulo the range of addresses accessed within the tile, as given by the footprint analysis. In the same way, only the store instructions and address calculations are preserved in the *store* phase, and load instructions are inserted to load the data to be stored from the scratchpad before the store occurs. Since all data is therefore stored at the end of the LES region, all data must also be loaded at the beginning of the region so that garbage is not written back to memory. To ensure that all data within the region is loaded and stored, both the *load* and *store* phases disregard any tile-internal control flow. In writing back all data at the end of the LES region, no additional storage space needs to be spent on keeping track of which data has been changed, e.g., "dirty bits" in caches.

Furthermore, if there are variables that are accessed in parent scheduling intervals, these are already available in the scratchpad, and must not be loaded again as this would overwrite changes done in the scratchpad that have not been written back yet. While it would not introduce any correctness issues to write back these values in the Store phase, it is not necessary as these values will be written back by the parent scheduling interval that originally loaded them.

In the *execute* phase, all loads and stores are transformed to access data from the scratchpad memory, as opposed to the original data in the DRAM. Thus, only the Load and Store phases will access the DRAM, and since these are executed in the PREM Memory phase, the PREM isolation property is respected. Lastly, as the Load and Store phases have been cleared of all non-data movement, the entering of child scheduling intervals, i.e., those of sub-loops, are always performed from the PREM compute phase, thus matching the synchronization stubs as presented in Section III-B2.

IV. EVALUATION

We implement HePREM as part of Clang-YKT [16], a fork of the LLVM-based C/C++ frontend Clang supporting OpenMP 4.x compilation for NVIDIA targets (PTX). As an evaluation platform we use the NVIDIA Tegra TX1 SoC, featuring ARM big.LITTLE A53/A57 executing NVIDIAs Linux4Tegra without modifications. Each *streaming multiprocessor* in the GPU feature a 48 KB scratchpad, and all scheduling intervals are selected to maximize the use of the scrachpad

[4]Of course the loop nest could be implemented as a *compatible interval*, as this scheme is not subject to the space limits of the scratchpad.

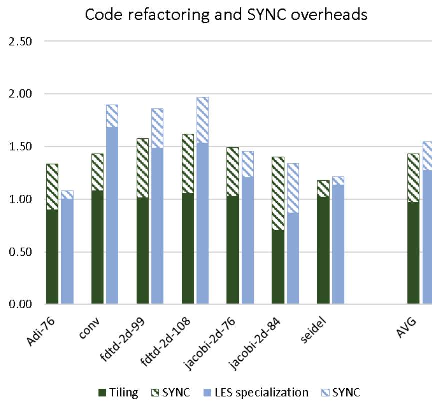
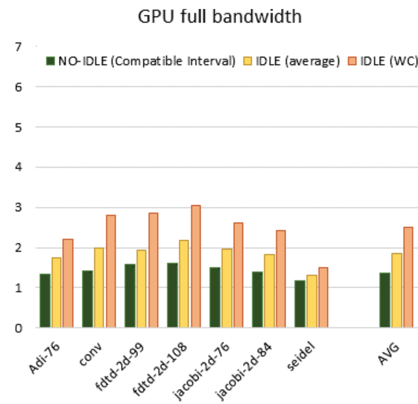


Fig. 4: Overheads of code refactoring and synchronization.



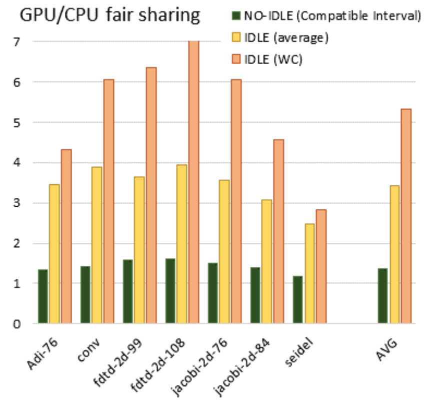


Fig. 5: Idle time introduced when budgeting for the worst case and average timer latencies under two different system memory schedules.

memory. Benchmarks are extracted from PolyBench-ACC [1], and all measurements are performed using *CUDA events* on the kernel offloading only.

In the following, we first describe the overheads implied by the HePREM compilation/synchronization and their implications on overall performance. Then, we discuss predictability (freedom from interference) results. As outlined previously, this paper focuses on GPU performance and predictability, with the CPU only acting as an interfering part.

A. HePREM Overheads

Figure 4 shows execution time of PREM-transformed code, normalized to the original program. We show a breakdown of code transformation and synchronization overhead for both *compatible intervals* and LES regions. It can be seen that *tiling* often decreases the execution time compared to the baseline, due to better cache locality. LES specialization adds around 25% overhead on average[5]. Synchronization with the CPU adds up to 50% overhead, in both cases.

PREM memory and compute phase lengths must be budgeted such that their WCET fits within the allocated length, as outlined in Section II. Due to the regular structure of GPU benchmarks and to our choice to use the scratchpad as local storage, the execution of the kernel itself has a low timing variance (i.e., budgeting for its WCET does not add much overhead). Linux timer interrupt latency, on the other hand, has a much higher variance. In our experiments the average timer interrupt latency is $5.6\mu s$, which can in some cases increase to $64\times$ the average[6]. To avoid the increase of GPU idleness, the periods for PREM memory and compute phases should be in line with the time required to fill and drain the scratchpad. Since the scratchpad is quite small (48KB), a high interrupt latency becomes comparable to the time spent on useful work, which ultimately implies inserting a lot of GPU idleness, when budgeting for the worst-case delay.

To study this effect, the benchmarks are executed in two configurations. First, we budget only for the *average* interrupt latency, knowing that this will lead to budget overruns when the interrupt latency is longer. Second, we budget for the *worst case* interrupt latency, thus ensuring that there will never be any budget overruns, at the cost of a large amount of idling in the common case.

Figure 5 shows the additional overheads on execution time of *compatible intervals* due to idleness insertion. We show two plots: to the left we consider giving the GPU the full compute bandwidth by producing a system-level schedule that optimizes the length of the GPU memory and compute phases exactly to the budgeted times. This is of course not a realistic schedule, as it would totally starve the CPU (*compatible intervals* have empty compute phase), but it clearly shows what can be achieved on the GPU side at best. To the right, we enforce a fair 50-50 time division between the CPU and GPU, which means that we introduce i) 50% idleness to the execution of *compatible intervals*; ii) additional idling in the shorter of PREM phases, as the memory and compute phase are now enforced to the same length.

Focusing on the plot to the left, the *compatible intervals* show a slowdown of nearly $2\times$ when budgeting for the average case and $2.5\times$ when budgeting for the worst-case. On top of that, when applying fair memory sharing (50-50) with the CPU the overheads due to idleness are roughly doubled. This is, of course, expected, as *compatible intervals* can perform no work during the compute phase. In the next section we discuss how execution of PREM LES code improves over that.

B. Co-scheduling CPU and GPU under interference

The main goal of HePREM is to ensure that the execution time of GPU programs remains the same under memory interference from other devices in the system. To evaluate these effects, we execute the original unmodified program as reference point, then we observe how the execution time of various schemes increases when we execute an interfering workload[7] on the CPU, under the fair sharing 50-50 scheduling. We compare results for the unmodified program and the three variants of the PREM-ized program introduces in Section II-A. For comparison we use the same tiling granularity, i.e., number of iterations per tile, for all versions. Furthermore, as strict predictability guarantees can only be given when the system is budgeted for the *worst-case* interrupt latencies, we focus our evaluation on this configuration.

[5] And up to 65%. Note that applying standard transformations based on polyhedral analysis [15] before our tiling could improve memory access pattern in most cases.

[6] NVIDIA recently released a new Linux kernel compatible with PREEMPT_RT [17], which should reduce the maximum interrupt latencies.

[7] We configure the `stress` test (people.seas.harvard.edu/~apw/stress) to generate large amounts of memory requests.

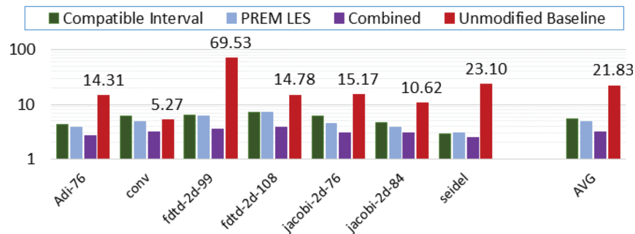


Fig. 6: Execution time of the four versions of the benchmarks under memory interference, normalized to the unmodified program executed in isolation.

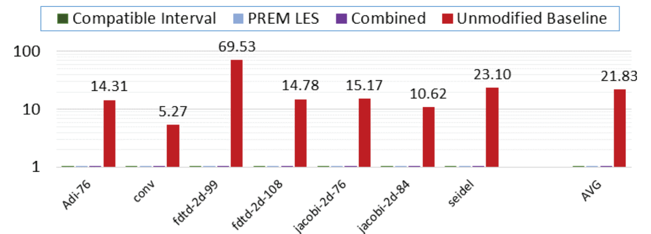


Fig. 7: Sensitivity to memory interference (the closer to one, the better).

Results are presented in Figure 6 (execution times normalized to the unmodified baseline without interference). There are three main take away points:

1) Performance of transformed programs: From Figure 4 it was clear that LES specialization has higher code generation overhead than *compatible intervals*. However, under 50-50 fair sharing system scheduling it is less susceptible to GPU idleness overheads. This is because LES does useful work during the *compute phase* on the data that has been pre-loaded to the local scratchpad, and thus the idling is reduced. The same is true for the combined schedule, with even higher improvements, as the amount of work per synchronization point is doubled. For some kernels, the *compatible interval* is better than LES, because the amount of work done within the compute phase is too small to amortize the additional cost of the specialization code. It has to be underlined that all the benchmarks considered here are fairly memory-bound. We expect the benefits of LES to be much higher for more compute-bound workloads. On average, LES is 12% faster than *compatible intervals* and *combined* is 35% faster than LES (and 42% faster than *compatible intervals*).

2) Performance improvement vs. unmodified baseline: As all PREM schemes provide freedom from interference, they all perform much better under memory interference than the unmodified baseline. While the latter on average slows down by more than $20\times$ when run under interference (compared to in isolation), the *combined* PREM schedule is only $2.5\times$ slower (due to the already discussed sources of overhead, not to the effects of interference). This is on average an order of magnitude improvement in the execution time under memory interference, with peaks of $70\times$ for *fdtd-2d-99*.

3) Freedom from interference: Figure 7 shows the sensitivity to memory interference of the different versions of the benchmarks. When budgeting for the worst case, all PREM schemes achieve essentially zero variance as desired, compared to up to $70\times$ for the unmodified baseline.

V. Conclusion and Future Work

PREM was first proposed as a method for executing real-time workloads on shared-memory, CPU-based systems with guaranteed freedom from memory interference. In this paper we propose a revisitation of the technique for heterogeneous on-chip systems. HePREM is composed of middleware for lightweight CPU/GPU synchronization, and compiler support to analyze high-level source code and create PREM-compatible regions that can be independently scheduled. We show that the compiled programs can achieve zero sensitivity to memory interference when the system schedule is properly budgeted, at the price of a relatively low overhead (25% code transform only, and 50% when synchronization is included, compared to baseline). While on average with our scheme we observe $2.5\times$ slowdown compared to the unmodified program executed in isolation, we achieve an order of magnitude speedup when running programs under interference. We are currently extending HePREM along several direction. First, we are extensively characterizing several workloads, to fully assess the benefits and limits of the technique. Second, while in the current work the focus is entirely on GPU execution, we are extending the compilation techniques to also transform CPU code, and the scheduling policies to produce a realistic system-wide distribution of target workloads. Finally, we are looking into techniques of extending support for code PREMization to library code in binary format.

VI. Acknowledgment

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

References

[1] S. Grauer-Gray *et al.*, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, 2012.
[2] 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.
[3] D. Dasari *et al.*, "A framework for memory contention analysis in multi-core platforms," *Real-Time Systems*, vol. 52, no. 3, May 2016.
[4] 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.
[5] 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.
[6] R. Pellizzoni *et al.*, "A predictable execution model for cots-based embedded systems," in *RTAS'11*. IEEE, 2011.
[7] A. Alhammad *et al.*, "Time-predictable execution of multithreaded applications on multicore systems," in *DATE'14*. IEEE, 2014.
[8] G. A. Elliott *et al.*, "Gpusync: A framework for real-time gpu management," in *2013 IEEE 34th Real-Time Systems Symp.*, 2013.
[9] Q. Chen *et al.*, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," in *ASPLOS'16*. ACM, 2016.
[10] P. Burgio *et al.*, "A memory-centric approach to enable timing-predictability within embedded many-core accelerators," in *RTEST'15*. IEEE, 2015.
[11] B. Forsberg *et al.*, "Gpuguard: Towards supporting a predictable execution model for heterogeneous soc," in *DATE'17*, 2017.
[12] M. R. Soliman *et al.*, "WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching," in *ECRTS'17*, 2017.
[13] 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, pp. 121–131.
[14] K.-A. Tran *et al.*, "Clairvoyance: Look-ahead compile-time scheduling," in *CGO'17*. IEEE, 2017, pp. 171–184.
[15] T. Grosser *et al.*, "Polly performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, 2012.
[16] S. F. Antao *et al.*, "Offloading support for openmp in clang and llvm," in *LLVM-HPC'16*, 2016.
[17] Real-time linux wiki. [Online]. Available: https://rt.wiki.kernel.org/index.php/Main_Page