Timing Predictable Execution for Heterogeneous Embedded Real-Time Systems

Diss. ETH No. 27390

Timing Predictable Execution for Heterogeneous Embedded Real-Time Systems

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

BJÖRN ALEXANDER FORSBERG
MSc in Engineering, Uppsala Universitet
born February 26, 1988
citizen of Sweden

accepted on the recommendation of

Prof. Dr. Luca Benini, examiner
Prof. Dr. Zdeněk Hanzálek, co-examiner
Prof. Dr. Andrea Acquaviva, co-examiner

2021

Acknowledgement

This thesis is the product of my doctoral studies under the supervision of Luca Benini, and I would like to thank him for the opportunity to pursue my doctoral degree and his great support. I would also like to thank my co-examiners Zdeněk Hanzálek and Andrea Acquaviva for reviewing this thesis and for the interesting discussions that have followed. My thanks also go out to Kai Lampka who initially steered me on the path to doctoral studies.

I would also like to thank my co-authors and collaborators, without whom this thesis would be significantly worse. Especially I would like to thank Andrea Marongiu for his support and guidance. My thanks also go out to Marco Solieri, Joel Matejka, Marko Bertogna, Michal Sojka, Andreas Kurth, Maxim Mattheeuws, Cyrill Burgener, Koen Wolters, Giuseppe Tagliavini, Alessandro Capotondi, and Thomas Benz. Thanks also to Frank K Gurkaynak and Jens Poulsen.

My thanks also to those with whom I have shared offices and coffee machines, the discussions with whom have made my time at ETH Zürich both more productive and more fun. In addition to the people mentioned above this includes Renzo Andri, Lukas Cavigelli, Daniele Palossi, Francesco Conti, Michael Herrsche, Antonio Libri, Mario Osta, Gianna Paulin, Matteo Spallanzani, and many others (you know who you are). I will remember the time we shared on the H- and J-floors of ETZ warmly! I extend the same thanks also to my flatmates Giovanni Volta and Helena Appelberg for making the 2020 lockdowns, during which this thesis was written, bearable.

Finally, I would like to extend my thanks to everyone who has given their support from back home in Sweden. Especially I would like to thank my family; Erik, Nina, and Sofia Forsberg. *Tusen tack!*

Abstract

The demand for computational power in real-time embedded systems has increased significantly, making multi-core and heterogeneous systems attractive in the real-time domain. However, as a single memory subsystem is shared by all cores, simultaneous use of the memory subsystem may significantly impact the timing properties of a task, and as systems must be dimensioned for their worst-case execution time (WCET), such *memory interference* may lead to very pessimistic execution times and low system utilization.

Simultaneously, another trend in real-time embedded systems is the increased interest in commercial-off-the-shelf (COTS) hardware, as it is cheaper and more performant than hardware platforms designed specifically with real-time timing guarantees in mind. Such systems are optimized for good average case performance, employing best-effort arbitration mechanisms that elevate the effects of memory interference of multi-core and heterogeneous systems.

In response to this, several software-based mechanisms to limit the effects of memory interference have been proposed. One of the most prominent is the Predictable Execution Model (PREM), which addresses the problem by dividing programs into sequences of memory and compute intensive *phases*, and scheduling the system such that the *memory phases* of two task never interfere with each other. Over the past decade a large body of PREM-compliant scheduling techniques have been proposed, however, very few works on how to automatize the laborious task of making programs PREM-compliant, and even fewer works that address the impact of architectural designs on PREM have been presented.

This thesis addresses this disconnect between scheduling and code and architectural considerations, starting with an exploration of implications to PREM of different system architectures, from multi-core CPUs, and via GPUs to Programmable Many-Core Accelerators (PMCA), as well as the impact of scratchpad- and cache-based memory hierarchies. From these results, we propose compiler-techniques to transform legacy code into PREM-compliant memory and compute *phases*, accomodating and optimizing for the different architecture and memory types. We show that such techniques can improve the performance of GPU kernels by up to 2×, but may incur a non-negligible

scheduling overhead determined by the refill-rate of the local memory, decided by its size. For CPU kernels freedom from interference transformations incur on average a 20% overhead, mainly due to cache management techniques. We also show that PREM can be done without significant overheads on PMCA, as PREM aligns well to the native execution model. We next confirm that the proposed techniques provide freedom from memory interference, showing that they reduce GPU execution time variance under memory interference by orders of magnitude to a few percent. Similarly, we show that the WCET of PREM workloads on CPUs can be up to 45% lower than traditional code. Finally, we show that inter-task optimizations, contrary to common belief, can not be well managed within the limited visibility of a PREM compiler, and propose an external optimizer toolchain that enable PREM systems to be optimized by trading task performance for overall memory performance. Using this technique we are able to reduce system response times by up to 31% over compiler-only techniques. We conclude that PREM requires different consideration for different architectural templates, but if well managed can provide *freedom from memory interference* guarantees over a vast array of different platforms, enabling timing-predictable excecution at low overhead.

Zusammenfassung

Die zunehmenden Anforderungen an die Rechenleistung eingebetteter Echtzeitsysteme haben Mehr- und Vielkernsystemen in den letzten Jahren zu einer attraktiven Lösung gemacht. Da aber alle Kerne in solchen Systemen sich das Speichersubsystem teilen, können parallele Speicherzugriffe mehrerer Kerne im Speichersubsystem interferieren und dadurch die Latenz von Echtzeitaufgaben erheblich beeinflussen. Zur gleichen Zeit hat das Interesse an Commercial-off-the-Shelf-Systemen (COTS) zugenommen, da sie im Vergleich zu auf Echtzeitgarantien spezialisierten Systemen günstiger sind und höhere Rechenleistungen bieten. COTS-Systeme sind allerdings für niedrige durchschnittliche Ausführungszeiten optimiert, und die dabei eingesetzten Best-Effort-Arbitrierungsmechanismen im Speichersubsystem können die erwähnte Speicherinterferenz verstärken. Echtzeitsysteme hingegen müssen für die Worst-Case-Ausführungszeit (WCET) dimensioniert sein, die durch Speicherinterferenz erhöht wird. Eine erhöhte WCET reduziert die Anzahl Aufgaben, die auf dem System verarbeitet werden können, und hebt dadurch den Hauptvorteil von Mehrkernsystemen auf. Eine zentrale Herausforderung bei der Einführung mehrkerniger COTS-Systeme für Echtzeitberechnungen ist deshalb die Minimierung der Speicherinterferenz.

Mehrere dafür geeignete Softwaretechniken wurden in den letzten Jahren vorgeschlagen. Einer der bekanntesten ist das *Predictable Execution Model* (PREM), das Echtzeitaufgaben in eine Sequenz von separaten speicher- und berechnungsintensiven *Phasen* aufteilt und einen Ablaufplan erstellt, in dem zwei speicherintensive Phasen nie parallel ausgeführt werden. Dadurch verhindert PREM das Auftreten der Umstände, bei denen Speicherinterferenz ein Risiko ist. Über das letzte Jahrzehnt wurde eine grosse Anzahl von Techniken für das Erstellen von PREM-konformen Ablaufplänen vorgeschlagen. Allerdings beherrschen nur wenige Techniken die automatische Transformation von Echtzeitaufgaben in Phasen. Eine noch geringere Anzahl der Techniken berücksichtigt bei den PREM-Transformationen auch die Besonderheiten der Rechnerarchitektur.

Diese Arbeit überbrückt die Lücke zwischen den Abläufplanen und dem automatischen und rechnerarchitekturbewussten Erstellen von PREM-konformem Code. Sie untersucht die Auswirkungen PREMs auf unterschiedliche Programme auf unterschiedlichen Rechnerarchitekturen, von Mehrkern-CPUs über GPUs zu vielkernigen Rechenbeschleunigern (PMCA). Auch die Implikationen von scratchpad- und cachebasierten Speicherhierarchien werden untersucht. Aus diesen Untersuchungen gehen Compilertechniken hervor, die Legacy-Code automatisch zu PREM-konformen speicher- und berechnungsintensiven Phasen, die für verschiedene Rechnerarchitekturen und Speicherhierarchien optimiert sind, transformiert. Die Leistungsbewertung zeigt, dass die präsentierten Lösungen GPU-Programme um 2× beschleunigen können, aber auch zu deutlichen Verlangsamungen wegen zusätzlicher Operationen durch das regelmässige Nachfüllen des lokalen Speichers führen können. Jedoch führen diese zusätzlichen Operationen bei CPUs, wegen ihrer grossen lokalen Speicher, zu einer durchschnittlich nur 20 % höheren Laufzeit, und können bei PMCAs, wegen der Übereinstimmung der Ausführungsmodelle von PREM und PMCAs, sogar komplett wegfallen. Die Bewertung der Effizienz der Techniken zeigt, dass die präsentierte PREM-Technik die Speicherinterferenz auf GPUs um mehrere Grössenordnungen und auf CPUs um bis zu 45 % auf nur wenige Prozent verringern kann. Zuletzt zeigt diese Arbeit, dass, im Gegensatz zu verbreiteten Annahmen, PREM-Systeme nicht von Compilern oder Ablaufplanern in Isolation optimiert werden können, und diese Arbeit präsentiert eine neues toolchainbasiertes Modell um diese Einschränkung aufzuheben. Diese Technik kann die Reaktionszeit eines Echtzeitsystems um bis zu 31 % im Vergleich zu nur-Compiler-Techniken senken.

Das Fazit ist, dass mit diesen Techniken das PREM automatisiert und mit geringem Zusatzaufwand auf eine breite Auswahl von COTS Mehr- und Vielkernsysteme angewandt werden kann, wodurch Speicherinterferenzen stark reduziert werden können.

Chapter 1

Introduction

The demand for computational power in real-time embedded systems has increased significantly, making multi-core and heterogeneous systems attractive in the real-time domain. However, as a single memory subsystem is shared by all cores, simultaneous use of the memory subsystem may significantly impact the timing properties of a task, and as systems must be dimensioned for their worst-case execution time (WCET), such *memory interference* may lead to very pessimistic execution times and low system utilization.

Simultaneously, another trend in real-time embedded systems is the increased interest in commercial-off-the-shelf (COTS) hardware, as it is cheaper and more performant than hardware platforms designed specifically with real-time timing guarantees in mind. Such systems are optimized for good average case performance, employing best-effort arbitration mechanisms that elevate the effects of memory interference of multi-core and heterogeneous systems.

1.1 The Memory Interference Problem

Over the past decade, multi-core systems have taken over every market segment, but their adoption is still slow in the context of real-time systems because contention on shared resources leads to unpredictable access times [1, 2, 3]. In recent years, there has similarly been a

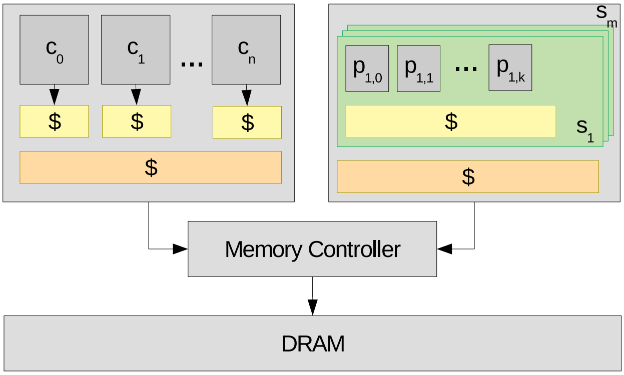


Figure 1.1: The architectural template considered throughout this thesis.

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 [4, 5].

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 all CPU cores and any accelerators in the system. This has large benefits in energy savings [6] due to reduced replication of power hungry hardware, and improves programmability, as programmers do not need to handle data movements between two discrete memories [7] when *offloading* computation to the accelerator.

1.1.1 System Model

Following this trend in embedded high-performance computing systems, the overarching architectural template considered in this work, as shown in Figure 1.1 consists of three parts. First, a CPU-like set of

N cores $c_0, \cdots, c_{N-1}$ which have at least one level of core-private storage. Second, one or more accelerator consisting of M clusters s_m of K cores, or processing elements, $p_{m,k}$. Third and last, a single shared memory system, as outlined in Section 1.1, to which all data and instruction requests from each of the CPU cores c_n and accelerator processing units $p_{m,k}$ are sent and handled. All of these components together are refered to as the *system*. Depending on the instantiation of a system, the point at which memory requests converge on their way to the memory may differ, i.e., at last level cache (LLC) level, memory controller (MC) level, or other part of the system.

With respect to the memory hierarchy, the only assumption made on the system is that each individual CPU core c_n and each accelerator cluster s_m have access to at least one level of private cache (highlighted in bright yellow in Figure 1.1). This is a necessary precondition for the application of the Predictable Execution Model (PREM), which is a cornerstone that this work builds upon. An introduction to PREM will be given in Section 1.3, after a discussion on the memory interference problem that this execution model is intended to solve.

On top of the system a number of real-time tasks $\tau \in T$, where T is the set of all tasks to be executed on the system, are deployed. Each task τ has an associated deadline before which its computation has to finish and the result returned. Consistent with traditional definitions of *hard real-time* systems [8], a failure of task τ, for any reason, to meet its deadline is considered a system failure. Each task τ is mapped either to a CPU core c or an accelerator cluster s, and for the purposes of this presentation we assume that there is no migration at runtime. While there is no fundamental limitation preventing migration, this problem is orthogonal to the focus of this thesis, and assuming a fixed task-core pairing keeps the discussion focused. We will return to a more precise task description in Section 1.4.

1.1.2 Memory Interference

To guarantee that timing constraints of tasks $\tau \in T$ are never violated, we assume that (and will partially explore how) the worst case execution time (WCET) of each task τ is analyzed to produce a *schedule* in which all tasks are guaranteed to finish before their *deadlines* [8]. On traditional single-core systems, such analysis is well understood and mature tools exist [9]. On the other hand, when multiple tasks are co-scheduled in multi- and many-core systems, they become susceptible to *interference* from each other's accesses to main memory (and from other peripherals' accesses), with significant impact on the WCET of real-time tasks [2, 3]. This contention induced by shared resources makes it difficult to bound worst case execution and response times, invalidating established single-core analytical methods for formal verification. Thus, for WCETs to provide a true upper bound under any multi- or many-core execution, the maximum interference would have to be assumed for every access [10], leading to very pessimistic bounds. These may even nullify the benefits of multi-core execution in the first place as memory latency increases.

The most severely contended resource is the global memory, e.g., the DRAM, from which all cores load instructions and data. This problem was shown by amongst others Pellizzoni et al [1] to be bad already in multi-core systems, where one could expect a linear increase in WCET with the number of cores added to the systems. Such findings have later been confirmed both on multi-core and heterogeneous systems by others, e.g., Caviocchioli et al [2] and Zhang et al [3].

Custom-designed hardware for real-time systems [11, 12], 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 defining software development guidelines aimed at enabling the long-awaited adoption of multi-core processors in safety-critical domains [13]. 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. This also enables *system composability*, an important property that ensures that adding or removing a software partition to or from the system does not affect the timing properties of any other partition, ensuring that the entire system does not have to be re-verified.

1.2 Software Techniques to Address the Memory Interference Problem

As is known from the literature [14], the deployment of high-level software arbitration mechanisms can provide a real-time aware abstraction layer which provides such *software partitioning* for *robustness to interference*. The abstraction layer removes all dependencies on unpredictable underlying hardware arbitrators by enforcing software implemented protocols that dictate which device that can access which resource at what time. Classical examples of such techniques include reservation server techniques [8], but during the multi- and many-core revolution this has been further extended in the last decade.

One class of techniques is based on the enforcement of per-core budgets. In these approaches, tasks are allowed to execute as long as they stay within the bounds of a predefined amount of cache misses. If this budget is exceeded, the task/core is stalled as to not negatively affect other tasks in the system. Examples of this include MemGuard [15], and BWLOCK [16]. While these approaches limit the amount of interference that different software partitions can have on each other, they are not able to eliminate it by design. This means that they remain incompatible with the vast amount of established single-core analytical methods, as some amount of external memory interference still needs to be considered in the timing correctness analysis of each partition. The aforementioned techniques are primarily intended for multi-core systems, and techniques for heterogeneous SoC management have started appearing more recently, and include scheduling of DMA memory transfers and kernel executions independently at offload time [17, 18], as well as an extension of BWLOCK – to BWLOCK++ [19] – to provide the capability of reserving memory bandwidth for offloaded kernels. Furthermore, SiGAMMA [20] similarly provides a reservation server, as well as techniques to interrupt misbehaving GPU kernels based on their memory bandwidth utilization.

Another class of techniques are cache-aware analytical methods for bounding the WCET under interference [21, 22, 23]. These techniques do not prevent interference, but attempt to tightly bound its effect to ensure that all tasks meet their deadlines without enforcing budgets.

The main drawback of these type of approaches is that the memory interference analysis needs to be redone each time a new task is added to the system, compromising the important property of *system composability*, i.e., that components can be added and removed from the system without affecting the parts of the system that have already been validated.

The third class of techniques [24, 25] is the enforcement of execution models that guarantee that segments of tasks in different partitions that require access to shared resources (memory) are isolated from each other in time through means of scheduling. This ensures that each individual segment can be analyzed with classical single-core analysis methods, and then combined into a system schedule without affecting these timing properties. In effect, this approach takes the software partitioning to an extreme, by subdividing individual tasks into separate, although communicating, partitions. One of the most prominent is the Predictable Execution Model (PREM) [24], which is the theoretical framework that underlies this thesis. The main goal of PREM is to remove interference from the system by design, meaning that separate tasks are no longer able to expose each other to memory interference, and the construction of real-time systems is simplified to the single-core equivalent state where system correctness can be guaranteed by finding a processor time *schedule* such that all tasks meet their deadlines. To achieve this, it has to be guaranteed that tasks can not affect each other's execution time through memory interference.

PREM achieves this by dividing programs into sequences of memory and compute intensive *phases*, and scheduling the system such that the *memory phases* of two task never interfere with each other. As such, PREM has mainly been considered as a scheduling approach over the past decade, with a large body of PREM-compliant scheduling techniques having been proposed [26, 27, 28, 29, 30, 31].

However, very few works [32, 33] have been published on how to achieve the fine-grained partitioning of tasks into separate phases, thereby automatizing the laborious task of making programs PREM-compliant, and even fewer works that explore how COTS hardware designs affect PREM [34] have been presented. This thesis addresses this disconnect between scheduling on the one hand and code and architectural considerations on the other.

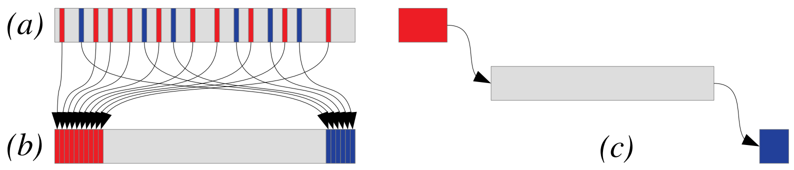


Figure 1.2: Schematic illustration on how PREM groups memory accesses spread out throughout a code segment (a) into coarser memory phases (b). This separation results in independently schedulable PREM phases (c).

1.3 The Predictable Execution Model

The Predictable Execution Model (PREM) was originally proposed in the context of single-core CPUs [24], 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 [28]. PREM [24] separates programs in scheduling intervals that can represent memory or compute phases, which will be discussed in detail in Section 1.3.1. *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.

1.3.1 The Three-Phase PREM Interval

The insight that underlies PREM is that any access that *hits* in the local memory does not depend on the shared resource, i.e., DRAM, and the worst case execution time (WCET) can not be influenced by external memory interference. For misses, isolation (no impact on WCET) can be achieved by reserving the memory system exclusively for the memory access. However, as cache hit analysis is difficult for individual accesses [35], and the mechanisms required to protect them are costly, it is infeasible to do this on a per-access granularity.

Instead, PREM divides the program into coarse-grained *intervals*, depicted in Figure 1.2. The original code segment (a) has memory loads (red) and stores (blue) spread out across the execution time (horizontal). By grouping these together (b) coarse enough units are created to enable individually schedulable load, execute, and store phases (c). Thus, the *load* (or *prefetch*) and *store* (or *writeback*) *memory phases* are responsible for staging the data of the *interval* through the local (private) memory, such that the *compute phase* is guaranteed to hit in the cache. Thus, costly protection of each individual access is replaced with the protection of the coarser *memory phases*. Each 3-tuple of *prefetch*, *compute*, and *writeback* phases is refered to as a PREM *interval*. To ensure that the prefetched data indeed leads to cache-hits only, each PREM interval must map to a region of code whose memory footprint is small enough to fit into the local memory, such as a private cache or SPM. The *compute* (or *execute*) phase can then operate on the local data without accessing the shared memory.

1.3.2 Compatible Intervals for Legacy Support

The original PREM proposal [24] acknowledges that some parts of a program cannot be transformed to adhere to the three-phase construct of a PREM *interval*, such as *syscalls*. For such cases, PREM introduces *compatible intervals*, which execute the legacy code as-is as a single *memory phase*. This way, any code can be made compatible with PREM. The downside is that *compatible intervals* require access to memory during their entire execution, despite only a limited share is devoted to memory accesses, leading to a less effective utilization of the memory bandwidth, as no other task can utilize the memory system in the meantime. To separate three-phase PREM intervals from single-phase *compatible intervals*, the former are referred to as *predictable intervals*. Note that within the PREM framework, both interval types can be executed in a timing-predictable way through mutually exclusive scheduling of any interval that accesses global memory.

1.4 Tasking and Scheduling Models

To achieve mutually exclusive memory accesses the Predictable Execution Model divides each task $\tau \in T$ into a sequence of *intervals* $I_\tau = \{i_0, i_1, \cdots, i_n\}$. Each interval i internally consists of independently schedulable prefetch (P), compute (C) and writeback (WB) phases, where the P and WB phases are referred to as the memory (M) phases. The memory phases are responsible for moving the data from the shared memory to a core-private memory λ which is not subject to interference, upon which the C phase computes. Importantly, this means that only the memory phases P and WB need to be scheduled with mutually exclusive memory access. To ensure that all data can be stored locally, the size of the data accessed within an interval $size(i)$ must be dimensioned such that it is smaller than the size of the local memory $size(\lambda)$, as shown in Equation 1.1.

$$\forall \tau \in T : \forall i \in I_\tau : size(i) < size(\lambda) \tag{1.1}$$

There exist multiple valid partitionings of a task τ into intervals I_τ, the selection of which is the task of the compiler, which we will present in Chapter 3, and optimize in Chapter 6.

1.4.1 PREM Scheduling

The original PREM paper [24] considered co-scheduling of a single CPU and I/O peripherals, but PREM has since been extended to address inter-core interference in COTS multi-core systems [26, 27, 28, 29, 30, 36]. While the scheduling question itself is out of scope, an overview of techniques in the literature is presented here for completeness. Following this, a generic PREM scheduling model that covers the fundamentals of all schedulers is formulated, introducing the symbols that will be used throughout the rest of the thesis.

Extending the work of the original PREM proposal by Pellizzoni [24], Bak et al [26] performed the first evaluation of under which scheduling policy PREM performs best, and determined that this was the least-laxity first with non-preemptive intervals. This evaluation was based on a large set of simulated workloads under several scheduling policies. While real systems were used to provide indications, no effects present in real systems are evaluated.

Furthermore, special considerations regarding the scheduling of systems based on different memory hierarchy types have been proposed. A large portion of scheduling work assumes the use of hardware managed caches [27, 28, 29, 31, 36], as these are ubiquitous in COTS systems, and allow the decoupling of the technique from the SPM buffer allocation problem. However, as caches can be subject to unpredictable replacement policies, PREM scheduling techniques have also been proposed for SPMs [30, 32], that do not suffer from these problems, as all data movement is managed from software.

Initial PREM work [24, 26, 27] considered only a single *memory* phase to prefetch data. Separate *prefetch* and *writeback* phases, i.e., the *three-phase PREM intervals* discussed above, were introduced by Alhammad et al [28], and motivated by the need to explicitly evict data at the end of each interval. These three-phase PREM intervals (*prefetch – compute – writeback*) are used by subsequent works [29, 30, 31, 36], and were described in detail in Section 1.3.1. The three-phase intervals are necessary for SPMs, because SPMs require data to be explicitly moved in and out by software, but this finer grained level of control is also useful for cache-based systems, through techniques such as preventive invalidation [37], as we will explore in Chapter 5.

PREM schedulers can further be categorized as preemptive [27, 31] or non-preemptive [29, 30, 36]. While an initial evaluation of different scheduling policies [26] concluded that a non-preemptive scheduling policy was best for PREM, they also concluded that it can cause priority inversion. We note that all preemptive PREM schedulers require mechanisms to ensure that prefetched data is not evicted by the time the task resumes. In practice, this requires the cache to be partitioned on a per-task basis, thereby decreasing interval sizes and increasing the context switch overhead, which we will discuss in Chapter 4. To achieve tight response times for PREM, both Bak et al [26] and Yao et al [27] recommend the promotion of memory phase priority, as compute phases can thereafter be scheduled without dependencies, as they do not need mutually exclusive access to memor, as compute phases can thereafter be scheduled without dependencies, as they do not need mutually exclusive access to memory.

1.4.2 Generic Model for PREM Scheduling

The objective of PREM scheduling is to ensure that *memory interference* is effectively avoided, while still ensuring that all tasks τ meet their deadlines D_τ. Memory interference is avoided by finding a system schedule that maps each interval i to a core c, and globally scheduling the system such that only a single core c is executing the memory phase of an interval i at a time. Scheduling techniques to achieve this are readily available in the literature [31, 29, 36], and as all share the fundamental requirement that only one task is executing its memory phase at once, the total response time R_τ of a task τ can be generically modeled as shown in Equation 1.2.

$$R_\tau = B^{core} + B^{memory} + S(|I_\tau|) + e_\tau \tag{1.2}$$

Here, B^{core} is the *blocking time* due to core-local scheduling, e.g., the increase in the response time due to τ being preempted (between intervals) by another task executing on the same core. The B^{memory} term is the *blocking time* due to a τ having to wait for a task on another core using the memory, due to the *mutually exclusive* policy at the heart of PREM. The S term is the static cost of performing the context switch for performing the online scheduling decision. This cost may vary from small (e.g., cost of a function call to determine the next interval in a pre-computed static schedule) to very large (e.g., a *syscall* and online decision from a dynamic scheduler). This cost grows linearly with the number of intervals $|I_\tau|$ in τ that require handling during execution [38]. The specific scheduling policy (e.g., fixed priority, earliest deadline first, etc.) determines when a task is blocked. Lastly, the e_τ term is the accumulated worst case execution time of all intervals in task τ, as shown in Equation 1.3.

$$e_\tau = \sum_{i \in I_\tau} len(i) \tag{1.3}$$

Here, $len(i)$ is the worst case execution time (WCET) of interval $i \in I_\tau$. For the remainder of this discussion, we will assume that $len(i)$ is provided by an external tool which we will refer to as the *WCET analyzer*, of which many have been proposed in the literature, as surveyed by Wilhelm et al [9]. As PREM scheduling implies *single-core equivalence* for the WCET analysis, classical single core analysis techniques can be used. As is customary, we say that a taskset T is schedulable if every task in the taskset responds before its deadline, as shown in Equation 1.4.

$$\forall \tau \in T : R_\tau < D_\tau \tag{1.4}$$

For the remainder of the discussion, we will only consider a single task τ executing per core c, and as such the term B^{core} will always be zero, assuming $|T| \leq N$. However, the fundamental insights of this paper generalize to the case where multiple tasks are deployed on each core, although the relative impact of B^{core} on the remaining terms may lead to a different optimal schedule. Following this, we revise Equation 1.2 as shown in Equation 1.5.

$$R_\tau = B^{memory} + S(|I_\tau|) + e_\tau \tag{1.5}$$

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

1.5 Contributions and Publications

As the main interest on PREM within the scientific community has been on developing efficient scheduling techniques (Section 1.4.1) for the three-phase interval model (Section 1.3.1), the evaluation of PREM has been limited to generated tasksets of different phase lengths, run in different forms of simulators.

This work presents the first exploration of the necessary software-support and the first exploration of the effects of the Predictable Execution Model when applied outside the confines of simulated multi-core CPU operating system to heterogeneous architectures with programmable accelerators, with a focus on CPU+GPU embedded platforms, such as the NVIDIA Jetson Series. By orchestrating the access to main memory between the CPU and GPU, as shown in Figure 1.3, it is possible to remove all sources of memory interference between the host processor and the accelerator. Chapter 2 provides the necessary

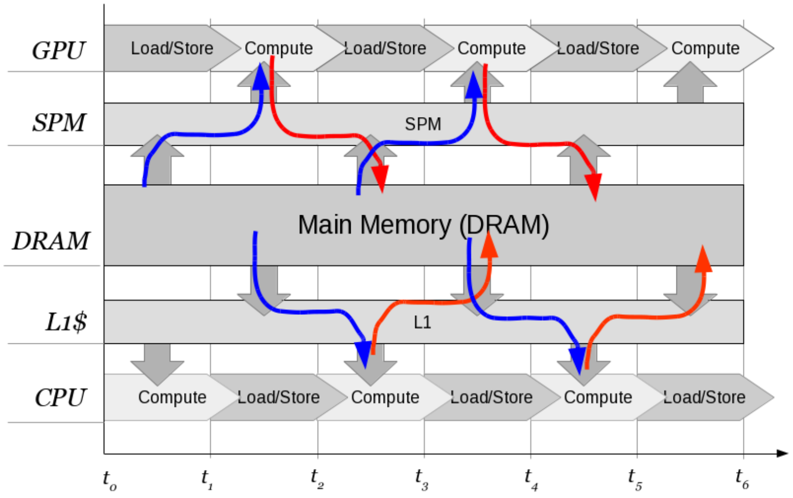


Figure 1.3: Heterogeneous PREM overview.

insights required to construct a PREM compiler capable of compiling heterogeneous programs for heterogeneous platforms. Chapter 3 presents the PREM compiler itself.

In Chapter 4 we use the presented PREM compiler to generate PREM-compatible versions of real workloads, and execute them on real systems, making a major contribution in establishing that the proposed techniques applicable to real systems, and quantify the impact that the necessary code transformations have on the achievable performance of real workloads. An important such system-application aspect is further presented in Chapter 5, which explores memory hierarchies from a predictability perspective, providing new insights on how caches in commercial systems can be used in a predictable manner.

Lastly, in Chapter 6, we automatize a design space exploration for PREM applications to tune the compiler to the platform an produce efficient systems. Due to the platform-specific tuning knobs, as well as inter- and intra-task memory optimizations that can be done, we conclude that an optimized PREM system can not be produced by an optimal PREM scheduler nor an optimal PREM compiler in isolation, but that optimization across all aspects of the final system are necessary.

The findings discussed in this thesis have primarily been presented as part of the following conference and journal publications:

- Björn Forsberg, Andrea Marongiu, and Luca Benini, "GPU-guard: Towards Supporting a Predictable Execution Model for Heterogeneous SoC," in *DATE'17*, 2017.
- Björn Forsberg, Luca Benini, and Andrea Marongiu, "HePREM: Enabling Predictable GPU Execution on Heterogeneous SoC," in *DATE'18*, 2018.
- Björn Forsberg, Luca Benini, and Andrea Marongiu, "Taming Data Caches for Predictable Execution on GPU-based SoCs," in *DATE'19*, 2019.
- Björn Forsberg, Luca Benini, and Andrea Marongiu, "HePrem: A Predictable Execution Model for GPU-based Heterogeneous SoCs," *IEEE Transactions on Computers*, 2020.
- Björn Forsberg, Maxim Mattheeuws, Andreas Kurth, Andrea Marongiu, and Luca Benini, "A Synergistic Approach to Predictable Compilation and Scheduling on Commodity Multi-Cores," in *LCTES'20.* 2020.
- Joel Matějka, Björn Forsberg, Michal Sojka, Zdeněk Hanzálek, Luca Benini, and Andrea Marongiu, "Combining PREM Compilation and ILP Scheduling for High-Performance and Predictable MPSoC Execution," in *PMAM'18*, 2018.
- Joel Matějka, Björn Forsberg, Michal Sojka, Premysl Sucha, Luca Benini, Andrea Marongiu, and Zdeněk Hanzálek, "Combining PREM Compilation and Static Scheduling for High-Performance and Predictable MPSoC Execution," *Parallel Computing*, 2019.
- Björn Forsberg, Luca Benini, and Andrea Marongiu, "On the Cost of Freedom from Interference in Heterogeneous SoCs," in *SCOPES'18*, 2018.

- Björn Forsberg, Daniele Palossi, Andrea Marongiu, and Luca Benini, "GPU-Accelerated Real-Time Path Planning and the Predictable Execution Model," in *ICCS'17*, 2017.

And the following manuscript, still in the publication process:

- Björn Forsberg, Marco Solieri, Marko Bertogna, Luca Benini, and Andrea Marongiu, "The Predictable Execution Model in Practice: Compiling Real Applications for COTS Hardware," submitted to *IEEE Transactions on Embedded Computing Systems.*

1.6 Outline

Following this introduction in Chapter 1, the thesis is divided into five chapters.

Beginning in Chapter 2 we explore techniques to extend the PREM scheduling beyond the CPU-resident OS scheduler to include all execution units of heterogeneous systems, with emphasis on GPU accelerators. To this end we present GPUguard, a synchronization infrastructure that enables scheduler control over GPU execution. This chapter proceeds with the manual transformation of a heterogeneous path planning application to conform to the requirements of PREM, and evaluate it together with GPUguard. From this we draw insights on the necessary steps, which leads to the development and exploration of compiler techniques for automatic *PREMization* in Chapter 3.

The compiler exploration is divided into two parts. First we directly use the insights from Chapter 2 to develop a PREM compiler for GPU applications. In doing so, we extend our insights from manual PREMization to compiler-based automatic PREMization, which is then further extended in the second half of Chapter 3 to address the question of more general applications, as are common on CPU systems. A thorough experimental evaluation of these compiler-based techniques follows in Chapter 4. We evaluate the techniques both with respect to the performance implications of the compiler transformations, and the ability of the transformed code to deliver on the

PREM guarantees on providing memory isolation through software partitioning.

The development of the compiler in Chapter 3 and the experimental evaluation in Chapter 4 lead to further insights on the interaction between the PREM execution model, and hardware features such as memory hierarchy configuration. In particular the impact of hardware-managed caches on the compiled PREM code is explored both in the context of CPU and GPU caches in Chapter 5.

Having explored the impact of PREM when combining a diverse set of benchmarks and systems, we commit Chapter 6 to discussing how PREM systems can be optimized. Fundamentally, the PREM schedulers presented in the literature over the past decade all assume that the PREM intervals are given constants that can not be influenced during scheduling. This may have been a reasonable assumption when changing the intervals required error-prone manual labour. However, with compiler-generated PREM intervals, their configuration can be arbitrarily changed by changing the compiler configuration – in turn opening up optimization opportunities that were not previously available. In Chapter 6 we explore the co-operation of PREM schedulers and PREM compilers to allow for dynamic resizing of PREM intervals to reduce blocking time in the generated schedules – thus producing better performing systems with shorter response times. We conclude with an overview of our findings, their impact, and future directions in the thesis conclusion in Chapter 7.

Chapter 2

Designing PREM for Heterogeneous Architectures

In this chapter we explore the requirements to implement PREM on a heterogeneous system, and perform an initial manual *PREMization* of a heterogeneous task. The main outcome of this exploration is to provide the necessary background for subsequently designing a PREM compiler, and the development of *GPUguard*, a synchronization-based technique to enable PREM scheduling to escape the confines of the CPU-resident OS. The manual PREMization is done on a GPU-based path planning algorithm, which then uses GPUguard to execute it predictably under memory interference on the NVIDIA Jetson TX1 heterogeneous CPU+GPU architecture.

This chapter provides the necessary information to develop the PREMizing compiler presented in Chapter 3, as well as the motivation for exploring the use of COTS caches with random replacement policies in connection to PREM presented in 5.

2.1 GPUguard: Extending PREM to Integrated GPU Accelerators

As accelerators supply a vast amount of computational power, typically even at smaller energy usage than CPUs, enabling their use within PREM systems is a critical step in meeting the computational demand for next-generation embedded real-time applications. Most modern high-end embedded SoCs rely on a heterogeneous design, coupling a general-purpose multi-core CPU to a massively parallel accelerator, typically a programmable GPU.

In such designs the coupling of CPU and GPU is very tight, as they physically share the main DRAM memory, as opposed to traditional *discrete* GPUs. As outlined in Section 1.1 main memory sharing complicates the deployment of real-time workloads, as memory interference may cause spikes in execution time that are difficult or even impossible to model and predict. This is particularly true in the view of the high bandwidth requirements of GPUs. To harness the advantages of COTS hardware and integrated accelerators in the context of real-time applications, new techniques that arbitrate memory requests are required.

In this section, we adress this issue from the perspective of embedded GPUs, in particular the NVIDIA brand. As will be discussed in Section 2.3, the insights gained from this exploration generalize to other forms of accelerators as well. This section describes the fundamental techniques necessary to achieve this, while the following section provides a deeper discussion on implementation details and an evaluation. Fundamentally, there are three issues that need to be addressed to achieve this.

First, to be able to leverage the previous work done in PREM sheduling, as discussed in Section 1.4.1, it is important that the specifics of accelerator execution be isolated and abstracted, such that the underlying differences in execution models (as opposed to CPU) do not impact the system and task models assumed in such previous work. This will be addressed in Section 2.1.1. Second, as the PREM scheduling can no longer be isolated to the CPU scheduler built into the operating system, a novel and portable way of managing the PREM phase scheduling across host-accelerator boundary needs to be identified. This will be addressed in Section 2.1.2. Third, a mechanism must be put in place that allows software control over the hardware-managed scheduling of GPU *warps*. A warp is the name given to the smallest unit of work schedulable by the GPU hardware, and typically consists of 32 threads executing in lock-step. As the GPU instruction set is specialized on computations, and it does not support any of the mechanisms used in general-purpose systems for scheduling, e.g., timer interrupts, another mechanism must be found. This is addressed in Section 2.1.3.

Following this, this section contains an evaluation of the OS and hardware impact on the presented approach. This section provides the fundamental techniques for applying PREM on a heterogeneous platform, which will be used in the following sections and chapters to enable further research into heterogeneous PREM.

Overall, this section describes the techniques and findings that were published in the *DATE'17* conference paper on *GPUguard* [39], and extended in 2020 with a manuscript in *IEEE Transactions on Computers* [40]. GPUguard enforces memory access isolation between tasks running on both the GPU and the CPU in a homogeneous architecture, where simultaneous accesses by several devices may cause spikes in the execution time and lead to overprovisioning of task time allocation because of varying worst case execution times (WCET). The goal of this work was to explore what mechanisms that are available to control the execution on the accelerator (in particular the GPU), and to integrate this with PREM execution.

In this work, the GPU scratchpad memory (CUDA shared memory) was used as local storage for PREM-like execution. This is the obvious choice, as the SPM is not subject to unpredictable cache replacement policies, which we will address in Chapter 5.

2.1.1 Execution Model Compatibility with Previous Work

Protecting the memory phases on the CPU is straight forward, as the OS scheduler has full control over which threads execute. On HeSoCs, the problem is more difficult, as the protection needs to be extended beyond the scope of the OS scheduler, to include the accelerators. To remain compatible with the scientific literature on PREM, the main requirement is that the GPU task adheres to the separation into memory and compute phases, as outlined in Section 1.3.1. By allowing the CPU scheduler to see the GPU phases as it would see a CPU phase, the GPU can be abstracted as a single classic PREM task, that when scheduled by the CPU scheduler acts as a proxy for the GPU memory phases. This retains the system-level scheduling with the CPU-scheduler, while allowing the GPU memory accessing phases to be controlled at a system level.

This implicitly leads to a PREM-compatible TDMA-style scheduling at system level. To provide guarantees on available memory access windows the lengths of the intervals are kept constant by enforcing the upper bound on the execution time of the phases. This is in line with previous PREM approaches. While double buffering is better for performance, single buffering provides the clearest division of memory and compute phases, and is the favored buffering technique for GPUguard.

As GPUs are executed in warps, this is the minimum granularity at which the GPU can be scheduled. However, as the GPU could potentially executing hundreds, or possibly thousands of warps at a time, it is not feasible to control the GPU at this level. Instead, GPUguard controls the GPU at a minimum granularity of Symmetric Multiprocessor (SM) level, which is NVIDIA terminology for a cluster. In our experiments we have not noticed any significant interference between different warps of the GPU if they are executing as part of the same task. As such, the selected approach is in line with the architectural design of GPUs, it is compatible with previous PREM approaches by high-level abstraction of the GPU as a task, and follows the generic system and tasking model as outlined in Section 1.4.

2.1.2 Memory Scheduling by Token Passing

To control the actual GPU execution from the proxy task on the CPU, the CPU needs to communicate with the GPU. As GPUs typically do not support any user-level controllable interrupts, or other forms of event-based synchronization schemes, GPUguard employs a portable memory-based synchronization scheme, by passing a *memory token* between the devices. Whichever device holds the token may access memory until the token is given up, as illustrated in Figure 2.1. On

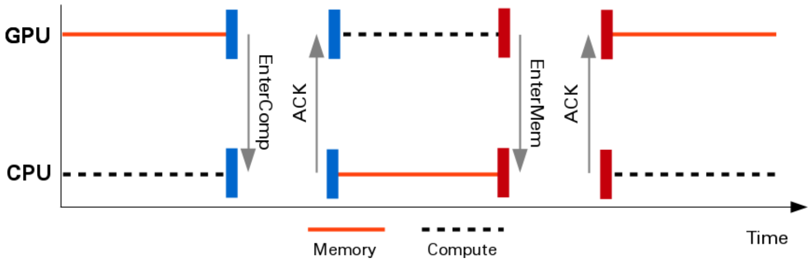
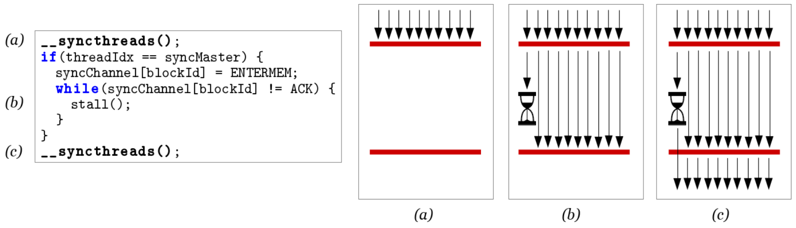


Figure 2.1: The synchronization protocol employed by GPUguard to change between computate and memory phases (phase lengths not to scale).

every phase change the GPU writes a sync flag into a segment of memory visible to both CPU and GPU, signalling if it is requesting to use the memory (*EnterMem*) or has finished using memory (*EnterComp*). Once the GPU has written the flag, it stalls until the flag has been unset, which signifies that the phase shift has been acknowledged (ACK) by the CPU. This ensures that the CPU is kept in control of when memory access is permitted. To ensure that the CPU can execute jobs in parallel, the CPU is not polling for the GPU sync flag, but only acts on the synchronization once the preset length of the GPU phase has passed, i.e., in line with classical scheduling time *quanta*, in this case determined by the WCET.

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 *quanta* associated with it, $E_{compute}$ and E_{memory}. In addition to this, the system schedule may delay the execution of the phases, e.g., due to memory being occupied by another task, which introduced idling I into the system. The quantity of I is determined completely by the exact schedule used for the system, and appears if the phase times T are shorter than the assigned E.

Synchronization is performed twice per PREM interval. Thus, taking the synchronization cost into account, which will be quantified in Section 2.1.4 the overall execution time of each interval $L_{interval}$ is



```
(a)  __syncthreads();
     if(threadIdx == syncMaster) {
       syncChannel[blockId] = ENTERMEM;
       while(syncChannel[blockId] != ACK) {
(b)      stall();
       }
     }
(c)  __syncthreads();
```

Figure 2.2: Implementation of stalling until sync on GPU. These synchronization points need to be encoded into the GPU kernel.

described by Equation 2.1.

$$L_{interval} = E_{memory} + E_{compute} + 2 \times S \tag{2.1}$$

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 Chapter 4.

As illustrated by Figure 2.1, this ensures that only one of the devices is accessing memory at a time. This approach trivially extends to control the GPU at a per-cluster granularity, by duplicating the synchronization *channel* through which the synchronization token is passed once for each cluster, enabling the CPU to control which GPU cluster s or CPU core c that is using memory at any given point in time.

2.1.3 Managing GPU Scheduling from Software

To ensure that the memory token is respected by every thread in a cluster, there must also be internal synchronization within each block on the GPU. Failing to do this may lead to only the thread that participates in the synchronization respecting the memory/compute phasing of the system.

As there is no event-based method, like interrupts, that can be employed within the GPU to affect control flow, the synchronization points for GPU kernels must be encoded within the program itself, explicitly before and after each PREM phase. We will discuss how this can be achieved manually in Section 2.2 and through compiler support in Chapter 3. To ensure that a consistent state (i.e., arrived at PREM phase boundary) can be communicated to the CPU at a per-cluster level, the GPU block[1] internally employs the synchronization scheme shown in Figure 2.2. At the boundary of each PREM phase, a *barrier* is inserted that ensures that every thread has finalized the execution of either the memory or compute phase, as illustrated by point (a) in the figure. Following this, one thread of each block (e.g., thread 0) will write the request for the memory token to the synchronization channel for the block that it is part of, and wait until the request is granted from the CPU. This is illustrated by point (b) in the figure. All other threads[2] will fall through to a second barrier waiting for the thread that is communicating with the CPU. Only when this thread reaches the barrier, all threads can continue into the next phase, as illustrated at point (c) in the figure.

This technique enables control of when GPU threads are executing the different PREM phases, even as thread scheduling on GPUs is otherwise managed from hardware and outside the reach of software mechanisms.

2.1.4 Implementation and Evaluation

This section provides an initial evaluation of GPUguard on the NVIDIA Tegra TX1 [41], a high-performance embedded heterogeneous architecture consisting of a 4-core ARM A57 host processor, and an NVIDIA Maxwell GPU. The goal of this initial evaluation is to quantify the different sources of overhead, and validate that GPUguard reduces the variability in memory access times.

PREM can be implemented at any level of the memory hierarchy. On the TX1 there are two main options available for the GPU, the SPM (CUDA *shared memory*) and the hardware-managed last-level cache (LLC). 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.

[1]A block is NVIDIA terminology for the unit of software executing on a cluster.

[2]More correctly, all threads that are not part of the warp to which the synchronizing thread belongs will continue, due to the *warp divergence* effect in GPUs.

GPUguard prototype

Here we implemented an evaluation prototype of GPUguard, split into a Linux loadable kernel module (LKM) and an API implemented in CUDA. This API abstracts the synchronization protocol as outlined in Sections 2.1.2 and 2.1.3. On the CPU, a call to the non-blocking HostSync() function starts the synchronization mechanism within the LKM. The non-blocking nature of the call allows the CPU to continue local execution while the kernel is running, as it would in any legacy CUDA program.

Since the CPU is tasked with ensuring that the CPU bandwidth is limited during GPU memory phases, the system must provide some mechanism to ensure that these limits are upheld. The encoding of the phases directly into GPU kernels address this issue at the GPU-side. Since the GPU is the focus of our evaluation, we do not PREMize the CPU application. Instead, to ensure that the CPU memory accesses are stopped during the GPU memory phase, the GPUguard synchronization mechanism is coupled with MemGuard [15], such that CPU tasks are stalled if they are missing in their cache[3] during the GPU memory phase. MemGuard uses LLC miss performance counters to detect when a core overruns its memory budget, at which point a high-priority real-time thread performing busy waiting is scheduled. This effectively preempts any running thread and ensuring that no further memory accesses are performed before the next renewal period. In this work we reuse certain aspects of the MemGuard mechanisms, such as bandwidth limitation through high-priority busy waiting. Once the GPU finishes its memory phase, the memory budget on the CPU is lifted.

Figure 2.3 shows the resulting setup. A user-space (top) program invokes the GPUguard prototype in the OS kernel space. It invokes the CUDA driver to start the execution on the GPU. The GPU execution, as shown on the bottom and in line with Figure 2.1, transitions between its memory and compute phases. At the end of each time quanta E_{memory}, a timer interrupt occurs which triggers the synchronization protocol described in Section 2.1.2, and enables

[3]While MemGuard allows for arbitrary thresholds of cache misses until the core is blocked, to achieve the full isolation guarantees of PREM, the CPU is throttled already at the first cache miss during a GPU memory phase.

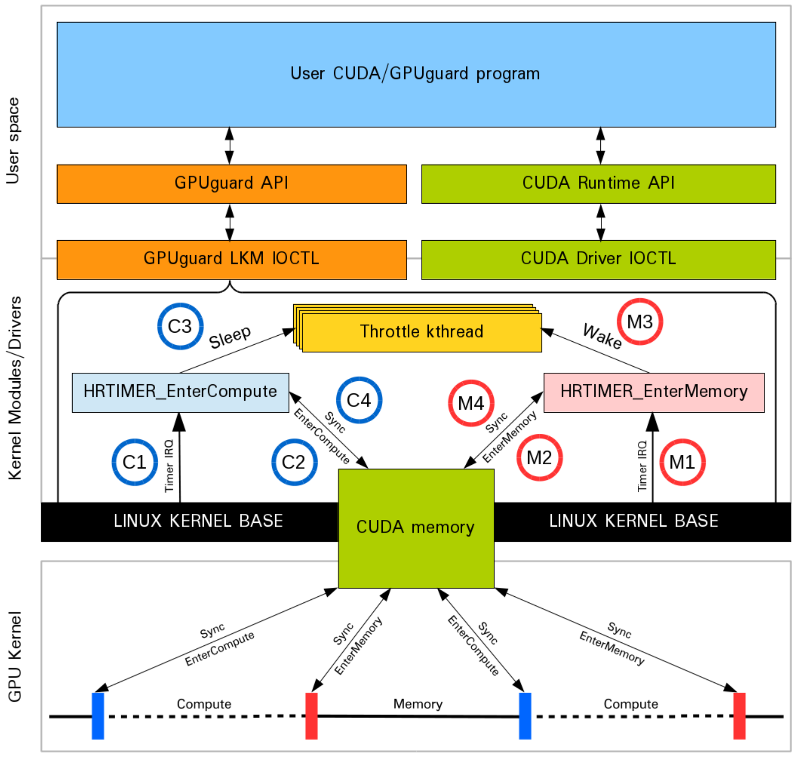


Figure 2.3: Overview of the GPUguard environment for the experimental evaluation.

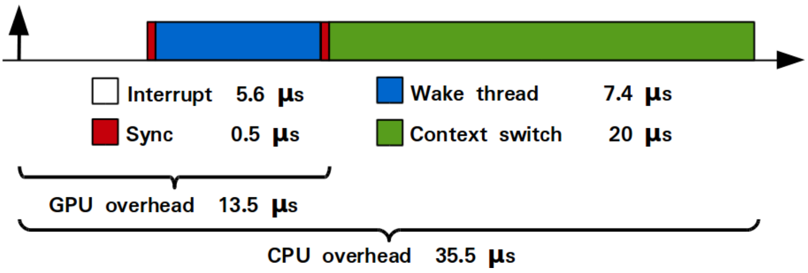


Figure 2.4: Measured latencies of the different steps involved in performing a EnterMemory synchronization.

or disables the MemGuard budget for the CPU. At the end of the $E_{compute}$ quanta, the synchronization protocol is invoked again, and the MemGuard budget disabled, enabling free memory access for the CPU. As such, it is guaranteed that the CPU cannot interfere with the GPU memory accesses while the GPU is in a memory phase.

OS and Hardware Characterization

To gain an understanding of the overheads imposed by the synchronization scheme, we have characterized the three main sources of overhead: First, the GPU synchronization overhead is the amount of time the GPU stalls while waiting for the synchronization acknowledgment from the CPU. Second, the CPU synchronization overhead is the degradation in performance experienced on the CPU due to the frequent interrupts induced by the synchronization. These overheads are represented by S in Section 2.1.2, here separated into S_{CPU} and S_{GPU} as the exact overhead depends on the device on which it is measured. Third, the period overhead is the cost of stalls due to bad matches between the GPUguard time quanta E and the actual execution time T of the GPU phases.

As an initial characterization of both the CPU and GPU synchronization overheads, the time required for all synchronization steps were measured. This includes the Linux timer interrupt latency (*interrupt*), the memory latency for writing the synchronization flags (*sync*), the time it takes to wake the throttle thread (*wake*), and

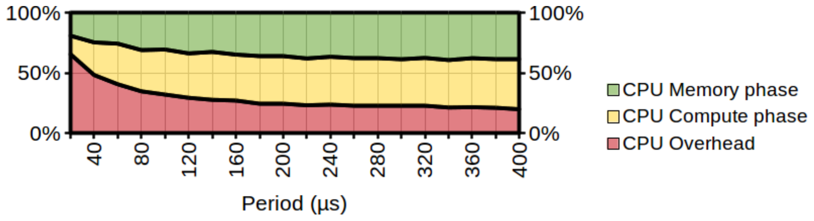


Figure 2.5: The percentage of the time available to CPU and GPU for useful work, as well as the overhead due to the synchronization.

the cost of context switching to the throttle thread once it has been scheduled (*context switch*). The results are presented in Figure 2.4.

To evaluate the CPU overhead, we created a synthetic benchmark which only performs synchronizations, and measured the available processing and memory time on the CPU. For this experiment, we assign 50% of the memory bandwidth to the CPU, corresponding to $\frac{E^{CPU}_{memory}}{E^{GPU}_{memory}} = 1$. However, as the results in Figure 2.5 shows, the actual un-throttled memory time available to the CPU is only about 40%, and decreases with the *period* length. This is because a shorter *period* implies more interrupts and more context switches, which is the source of CPU overhead S_{CPU}[4].

The GPU overhead S_{GPU}, presented in Figure 2.4, is smaller than that of the CPU, as the GPU does not perform any additional operations once the synchronization is done. However, it has to wait for the CPU to wake the throttle thread and acknowledge the synchronization request.

The final source of overhead is due to the over-dimensioning of the *GPUguard* quanta E compared to the phase lengths T. This overhead

[4]As *GPUguard* only throttles the memory bandwidth, a compute-intensive CPU task can continue execution during the entire *period*, as long as it does not require main memory access. On the other end of the spectrum, a memory-intensive CPU task will be subject to bandwidth throttling during the CPU compute phase, and in the extreme case it will be stalled until the next CPU memory phase. Thus, depending on the *compute-to-communication ratio* of the CPU tasks, and the bandwidth limit put in place through the *MemGuard* throttling mechanism, the available time for performing useful work for a specific task will be somewhere between these two extremes.

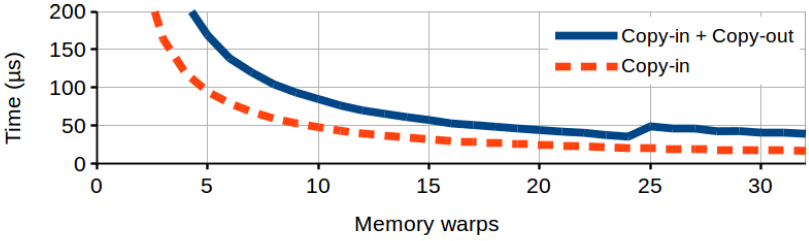


Figure 2.6: The time required to fill, and fill + empty, the scratchpad memory on the GPU using different numbers of memory warps.

appears when the *GPUguard period* is longer than the actual execution time of the GPU phases, causing the GPU to stall while waiting for the CPU to enter the synchronization phase. To gain an insight into the optimal length for E_{memory}, at which no stall occurs, we measured the time required to populate the scratchpad memory using different amount of memory warps. As PREM requires a synchronization each time the scratchpad is refilled, this is equivalent to a PREM interval sized to the local memory $size(\lambda)$ but with zero computation, and represents the worst-case compute-to-communaction ratio of the typical PREM interval size. We also measured the time to both populate and evict the scratchpad data, as this constitutes typical program behavior. For these measurements we use a synthetic benchmark which performs fully coalesced memory accesses to refill the scratchpad, which gives a lower-bound for the resulting memory phase lengths, as this is the most efficient way to access memory [42, 43]. The measured times are presented in Fig. 2.6. For most warp configurations, the time required to fill the shared memory is in the range where the CPU-side experiences heavy degradation due to synchronization overhead, see Figure 2.5. As a result of this, for kernels that have short phase lengths, the scheduling quanta E will have to be set higher than the actual T of the PREM phases to ensure that the CPU is not starved. Unless the compute phase is long enough to amortize this overhead, these overheads will become noticable. In effect, memory bound programs, i.e., programs that perform extensive searching or data traversal in relation to the length of the computation phase are susceptible to stalls. This will be discussed in

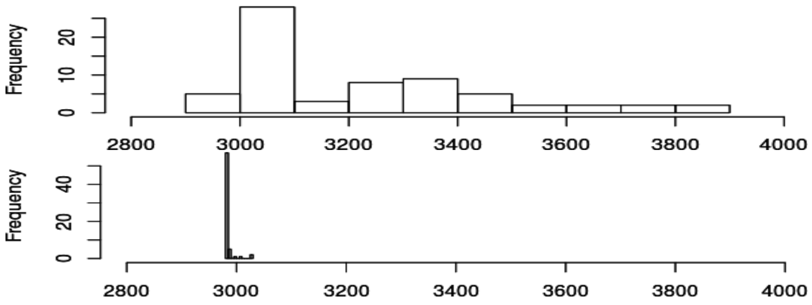


Figure 2.7: The distribution of execution times for the Matrix Multiplication kernel under contention. The top histogram shows the variance in the baseline CUDA version, and the bottom one shows the variance when *GPUguard* is used to orchestrate memory accesses.

greater detail in Section 2.2 where we PREMize a GPU-based path planning application, and Chapter 4, where we evaluate a large set of benchmarks of different compute to communication ratios. The main take-away from this experiment is that creating PREM intervals sized as close to $size(\lambda)$ is going to be important to make T large enough to allow the scheduling quanta E to be set $E = T$ to prevent stalls, and produce well-performing heterogeneous PREM systems.

Effects on Interference

While there are programs that will suffer significant overhead due to idling $I > 0$ or $S \gg E$, many classical GPU applications, such as BLAS (Basic Linear Algebra Subroutine) kernels, are compute intensive, and thus not affected by this limitation. We therefore employ a compute heavy program, matrix multiplication, a BLAS3-type kernel for the initial evaluation of *GPUguard*.

We implement a *GPUguard* enabled version of matrix-matrix multiplication and compare it to a baseline version taken from the CUDA samples provided by NVIDIA [44]. Matrix Multiplication has a compute to communication ratio of $2n^3/3n^2$, i.e., the computation part is cubic in the input size, while the communication is only quadratic. Thus, as the input size increases, the computational work increases asymptotically faster than the memory requirements, leading to less frequent synchronizations due to the scratchpad memory size. Furthermore, we set the CPU bandwidth limit to zero during the GPU memory phases, thus stalling all memory accesses from the CPU until the end of the phase. This represents the maximum achievable isolation – and is in accordance with PREM – and the best case for increasing timing predictability.

We execute the NVIDIA reference version and the *GPUguard*-enabled kernels over several iterations under memory contention generated by the CPU, and plot their execution time distributions in Figure 2.7. As can be seen in the figure, the execution time variance in the *GPUguard* enabled versions of the BLAS kernels is near zero, in contrast to the non-PREM reference implementation. In addition to this, the execution time of the *GPUguard*-enabled kernel is within the range of execution times exhibited by the CUDA reference implementation, showing that heterogeneous PREM execution is possible without introducing significant overheads from the sources discussed above.

2.2 GPUguard and Predictable Path Planning

Having understood the fundamentals of GPUguard, we now set to apply it to a real application, that fits into the set of compute-intensive workloads that require the computing power of modern heterogeneous architectures, as outlined in Chapter 1. Two of the most representative examples of such workloads are unmanned aerial vehicles (UAVs) [45] and autonomous driving systems [46]. UAVs are already used for tasks such as aerial mapping, entertainment, surveillance, and rescue missions. Fully autonomous driving is still out of reach, but virtually every major OEM has a roadmap towards achieving this goal, and is already commercializing advanced driver assistance systems (ADAS) [47].

In light of this, this section applies PREM and GPUguard to a GPU-enabled path planning algorithm – an important component in both of these systems – to improve its timing predictability. This section presents the work [48] presented at *ALCHEMY'17* hosted at *ETH Zürich*, and done in close collaboration with *Daniele Palossi* who provided the tracker implementation.

2.2.1 Preliminaries

To turn the path planner into a PREM-compliant form, the application must be modified to support the separated memory and computation phases, and integrated with GPUguard as presented in the previous section. We begin by presenting the non-PREM reference version of the path planner, and then present how we achieve this transformation.

Reference Path Planner

The reference path planner [49] has been chosen due to its good performance and accuracy, achieved through a non-deterministic implementation that trades a small amount of accuracy loss (1.2% on average) for a great performance gain (up to 3.7×). It builds upon a two-step process. First, *automata synchronous composition* [50], in which a discretize topology of the environment, represented as a graph, is merged with a second graph representing the kinematics of a robot, to create a *composition automaton* graph. If a location of the map is occupied by an obstacle, the corresponding node is removed from the map before the composition takes place. The main benefit of this approach is that the path returned by the path planner is guaranteed to be compliant with maneuvers that the robot is able to perform, simplifying post-validation of the path. The price for this is an increased size of the graph to be explored. In the reference implementation, the composition automaton is 21× larger than the map size. Once the composition automaton has been created, the final step in the process is to explore it using a Single Source Shortest Path algorithmm such as Dijkstra's algorithm [51]. The vertices to be explored in each iteration are referred to as *reference nodes*, and it has edges to a set of *neighbor* vertices. The main data structure is a sparse *state-transition matrix* used to represent all the vertices and the connecting edges. The information about which nodes are "to be visited" is kept in an auxiliary array called *mask array*, and the cost to traverse each node is stored in the *cost array*. While the mask array indicates that there are still vertices to be explored, another parallel iteration is performed, following Dijkstra's algorithm.

As this is a non-trivial problem to solve in real-time for a safety-critical system, such as an unmanned aerial vehicle, this is a clear candidate for deployment on modern heterogeneous embedded systems, as well as a clear candidate for predictable exceution techniques such as PREM, that can be enabled with techniques such as GPUguard.

This path planner is used as reference implementation. We port it from OpenCL to CUDA, first to a naive copy of the original, and then to an optimized version considering the target platform. For each of these two version, we implement a standard port, as well as a GPUguard-enabled version. Due to the large size of the maps, the graph representations are too large to fit into the GPU local memory. Thus, in order to handle such limitation, we need to introduce memory partitioning techniques, i.e., tiling or blocking techniques [52]. Indeed, this will be an important step for any program that has a larger footprint than the size of the local memory $size(\lambda)$ as we will further discuss in the next chapter, when using the experience of this transformation to construct a PREM-enabling compiler.

Code Adaptation for PREM: Warp Specialization

PREM requires code to be refactored into *prefetch*, *compute*, and *writeback* phases. In the GPU setting, an interesting work presented in the literature, CudaDMA [53], proposed a method of performing such a division to stage data through the local scratchpad through a process called *Warp Specialization*. In a warp specialized kernel, a subset of the threads are continously executing memory prefetches, while remaining threads perform the computation on the data available locally. An abstract overview of this is presented in Figure 2.8. In the original implementation, all load/stores were performed on the DRAM (1). After the program has been modified, the memory phase is bringing data in (2) and out (3) of the scratchpad. The computation phase is operating on the local data in the scratchpad (4). The entering of these phases is protected by barriers, which ensure that all copy operations or computations respectively have been completed. This work focuses on the GPU side of the execution, and thus the

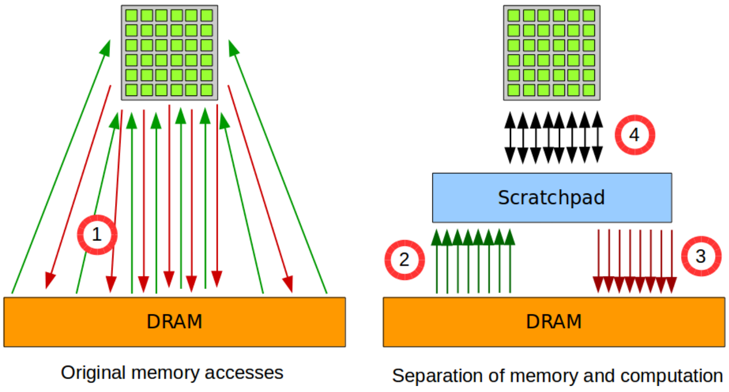


Figure 2.8: A visualization on how the memory access pattern changes in the phase separation of the GPU applications. In the original program, all memory accesses were done directly to the DRAM (1). After the modifications, the program is executed in distinct memory (2, 3) and compute (4) phases, where only the memory phase accesses the system DRAM.

CPU side is not further explored in this context.

While this was not originally intended for PREM or similar approaches, this provided the means to and end to achieve this on the GPU. The original motivation for CudaDMA and warp specialization was to achieve DMA-like behavior on NVIDIA GPUs that did not (and still do not) supply a hardware DMA engine. By emulating this in software, it has been successfully applied to improve performance for compute intensive programs with high data-reuse, thereby avoiding the refetching of data from global memory. In such cases, scratchpad data staging leads to performance improvements.

A key property of warp specialization is the assignment of different warps (groups of threads) to perform the memory and compute operations. In PREM terms, this enables the assignment of different number of threads to the compute and memory phases to change the duration of the memory and compute phases. This independent allocation of threads to perform computation or memory operations provide a mechanism to balance the length of the corresponding phases.

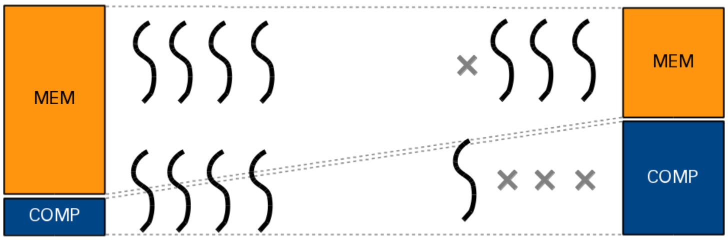


Figure 2.9: A visualization on how the assignment of memory and compute threads in a warp specialized program can lead to better balancing of compute and memory phases. In the naive case, the same threads are used to perform both memory and compute operations, but in the second case, the threads are allocated to balance the length of the two phases.

A visual representation of this is presented in Figure 2.9. By assigning a larger portion of the threads to memory than computation, the intuition is that phase durations can be balanced, reducing the stall time between the phases. In other words, GPU resources can be allocated in such a way as to provide scheduling quanta that support the natural phase division of concurrently running GPU and CPU tasks. This follows from the discussion in the original presentation of warp specialization [53], but is here used for the first time.

A detailed description of how GPU code can be transformed to warp specialized state is presented by Bauer et al [53], however, the fundamental steps are outlined here. First, the GPU kernel is divided into two parts by an *if-else* statement, which based on the thread identifier, *threadIdx*, to determine if the thread is a compute (*if*) or memory (*else*) thread. For the compute threads, the code is implemented to first wait on a *data-avail* barrier, after which computation is adapted to perform the original computation on scratchpad buffers. Once the computation on the data in the buffer is complete, it triggers a synchronization on a *data-refill* barrier. The memory threads perform the opposite operation, initially they wait on a *data-refill* barrier, after which they perform DMA-like data copying to writeback old buffer data into DRAM, and refill the SPM buffers with data for the next computation. Once this operation is finished, the memory threads trigger the *data-avail* synchronization. By utilizing multiple buffers, this technique can be extended to perform double buffering.

This structure is compatible with the *GPUguard* synchronization scheme as outlined in Section 2.1.3, by replacing the *data-avail* barrier with the *EnterCompute* synchronization, and the *data-refill* barrier with the *EnterMemory* synchronization. Therefore, as warp specialization both showed great promise and was compatible with the *GPUguard* synchronization protocol[5], it was used as an intermediate step for porting the tracker. However, the potential downsides of this approach when applied more broadly will be discussed in Chapter 4.

2.2.2 Six Tracker Implementations

Using the reference path planner presented in Section 2.2.1 two main versions, each in three flavors are produced. The first version is the *Naive* port of the code from OpenCL to CUDA, but without far-reaching optimizations for the NVIDIA platform. The second version is the *Coal* version, which introduces such optimizations to improve performance.

Naive Port of Path Planner

The *naive* port of the reference path planner [49] is produced in three flavors, the first – *Naive* being the direct CUDA port of the tracker from OpenCL without CUDA-specific optimizations. The second – *Naive-WS* – is the warp specialized version of the same code, which is created by applying the methodology in Section 2.2.1 to produce memory- and compute-phases with barrier synchronizations. The third version – *Naive-PREM* is the PREM-enabled version, which is created by exchanging the CUDA barriers of *Naive-WS* with the GPUguard synchronization primitives, for integration with GPUguard presented in Section 2.1.

However, as we will see soon, the *Naive* port is underperforming on the NVIDIA platform. The main issue is that the *transition matrix* which specifies the neighboring vertices in the graph is stored in a large matrix. As is common in graph-processing algorithms, this leads to

[5]Note that warp specialization is just one of many techniques that have this property.

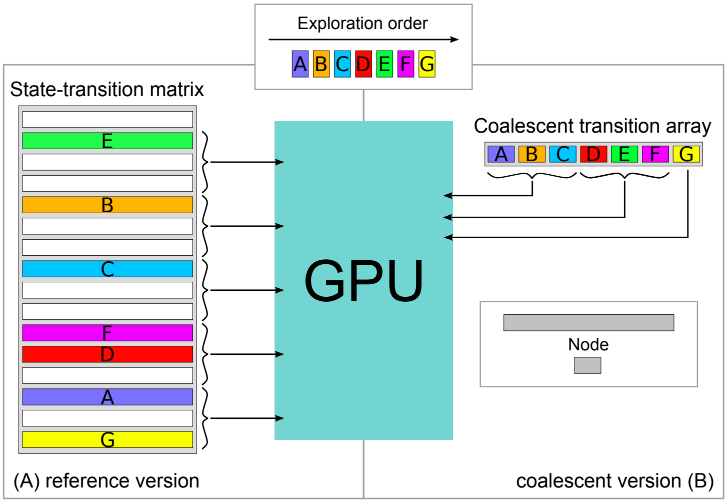


Figure 2.10: A visualization on the memory access pattern of the *Naive* (A) and *Coal* (B) implementations.

un-coalesced access patterns, as the graph nodes visited can be stored in completely different place in memory, as shown in Figure 2.10a. Furthermore, the store order of the *cost array*, indexed by the vertex ID, does not reflect the order in which the vertices are accessed, again leading to un-coalesced accesses as the cost accessed from each thread resides in different parts of memory.

Optimized Path Planner with Coalesced accesses

To overcome the poor memory performance of the *Naive* port, a pre-processing stage is introduced. This stage performs an offline exploration of the empty map, reordering the elements of the transition matrix such that they come in the order that they are explored by the sequential version. This change enables the streaming of the transition matrix to the GPU, which implies coalesced memory accesses and maximum use of the memory bandwidth. As some vertices of the graph may be explored multiple times, to keep the streaming property of the transition matrix, these vertices must be added multiple

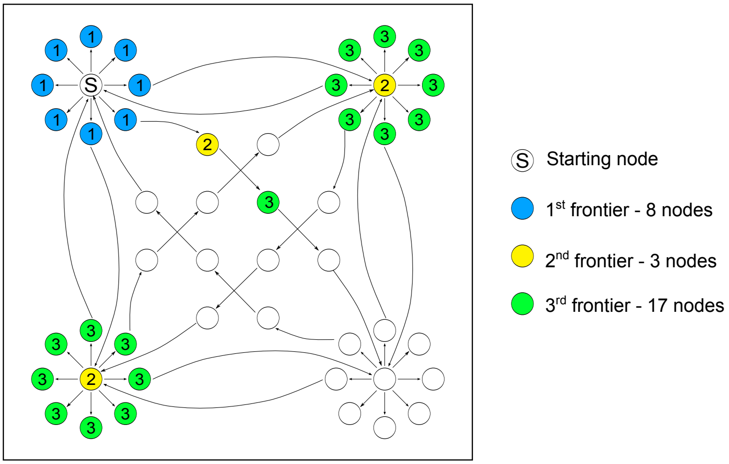


Figure 2.11: A example on how the frontiers are constructed in *Coal*.

times. We refer to this new version as *Coal*. A visual representation of its access pattern is presented in Figure 2.10b, which shows that each memory access now brings in multiple vertices that are to be explored. As the cost array is updated by multiple nodes, storing it in the visit order would introduce coherency issues due to the duplication, thus it is kept in the original format.

The calculation of the cost to reach each node is greedy, as inherited from the original Dijkstra implementation for the path planner. For each node the neighbors are explored, and if the cost to reach the neighbor from the current reference node is lower than the previous cost, the cost is updated, and the predecessor of the neighbor is set to the reference node. Thus, the algorithm breaks if a thread explores a node which has not yet had its cost updated, as this error would propagate to all its successors. Especially on a GPU, where hundreds or thousands of nodes could be explored at once, a mechanism which prevents this must be implemented.

In the *Naive* implementation, this was addressed using the mask array, but for the streaming transition matrix, this is no longer feasible. Instead, the concept of *exploration frontiers* is introduced. The exploration frontiers is an enumeration of sets of vertices F, where all vertices in F_n have been visited from at least one vertex in F_m for any $m : 0 \geq m < n$. As shown in Figure 2.11, the base case is F_0 which contains only the source vertex. The next frontier is constructed by all the vertices that can be reached from the source vertex, and then the remaining frontiers are in turn populated by the vertices that can be reached by the previous frontier. The introduction of the frontier concept enables the insertion of breakpoints in the streaming transition matrix, at which point all previous vertices have to have been explored before the exploration can continue beyond that point in the stream, thus ensuring that nodes are not visited out of order. All of these operations are done offline and encoded into the streaming transition matrix.

This consitutes the base version for *Coal*, and like for the *Naive* port we also create a warp specialized flavor – *Coal-WS* – and a PREM-enabled flavor – *Coal-PREM* – using the same methodology. As the focus here is on PREM, further details of the algorithmic details are out of scope, but further details on the implementation and accuracy of the algorithm are presented by Palossi et al at *CF'16* [49] and *SCOPES'17* [54].

2.2.3 PREM Evaluation

Using the presented implementations of the planner, we evaluate them from a performance and predictability perspective. The performance evaluation will tell us important information about the impact of the code transformations (i.e., warp specialization and *GPUguard* synchronization) to the performance of the code. The predictability evaluation allows the exploration of how the execution time stability is affected by memory interference for the different versions, and provides a direct measure on the effectiveness of GPUguard and PREM to reach these goals.

For this purpose we use the NVIDIA Tegra TX1 [41] heterogeneous SoC which features both a 4-core ARM Cortex A57 and an onchip programmable NVIDIA Maxwell GPU. The GPU consists of two streaming multiprocessors (SM), each capable of executing 4 warps of 32 threads concurrently on the 4 × 32 GPU cores. The threads of a warp are executing in lock-step, i.e., sharing the same program counter. Logically, programs are divided into blocks of threads, where

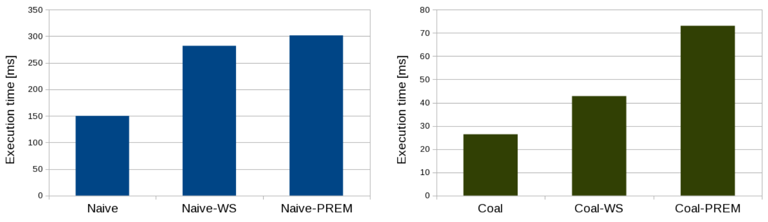


Figure 2.12: The absolute execution times for the naive (left) and coalesced (right) implementation of the path planner. Note the different ranges of the Y-axes.

each block can have a maximum of 1024 threads executing at a time. Each GPU core has a private L1 cache, and all threads executing within the same block, i.e., on the same SM have access to a shared L2 cache and a 48 KB scratchpad memory.

We evaluate the implementation on map sizes of 100×100, as this is the maximum size considered in the original paper [49]. All versions are executed with 1024 GPU threads (i.e., 32 warps, see Section 2.1.4). For the PREM-enabled versions, we configure the *GPUguard* scheduling quantas to match the empirically found WCET of each phase when executed without memory interference, i.e., $E = T$, and throttle the CPU bandwidth during GPU memory phases. The base and warp specialized versions do not interact with *GPUguard*.

Performance

The execution times for the six path planner versions presented in the previous section are shown in Figure 2.12. The first thing to notice is that the *Coal* versions are $6\times$ faster than the *Naive* ports (notice different Y-axis scales). This is due to the changes to the data structures that enable coalesced memory accesses, making full use of the GPU bandwidth.

The performance loss for implementing warp specialization for the *Naive* port (i.e., the difference between *Naive* and *Naive-WS*) is significant, almost $2\times$. GPUs are known for hiding memory latency by having many in-flight requests at a time. In the original version, memory requests were in-flight during computation, but due to the barrier synchronization, no memory accesses are in-flight during the compute phase of *Naive-WS*. Therefore, the full memory latency has to be paid at the start of each memory phase, i.e., a cold start penalty. This effect is somewhat amortized in *Coal*, where *Coal-WS* adds less overhead. This is due to *Coal* being less reliant on having a large amount of in-flight requests, as each request on average has a lower latency, due to coalesced access patterns.

When moving from the warp specialized versions to the PREM-enabled, the execution time further increases. This is expected, as the GPUguard synchronizations with the CPU are bound to require more time than the barriers used in the *WS* versions. However, there is a big difference in the size of the overhead between the *Naive* and *Coal* implementations, in the former the overhead is marginal, while the execution time almost doubles for *Coal-PREM*.

The reason for the difference is twofold. The first reason is that the *GPUguard* synchronization cost with the host is fixed in size, and how well it is amortized by useful work dictates how much execution time overhead it adds, as outlined in Section 2.1.2. In the case of *Naive*, with its unoptimized memory accesses, the memory phase is significantly longer than for the *Coal* version, meaning that the overhead is better amortized in *Naive*. This effect will be further explored in Section 5.1.1.

The other effect is due to algorithmic properties of *Coal* interacting with *GPUguard*. The *Coal* implementation stops loading vertices when it encounters a frontier breakpoint. In iterations where the breakpoint does not coincide with the SPM being full, this leads to shorter durations of memory and compute phases. However, as *GPUguard* is configured to always enforce the WCET of each phase, through the scheduling quanta, the execution of a near-empty iteration will require the same time as a full one – this is not the case for the non-PREM versions of *Coal*. In the tested map configuration this occurs frequently, but could be addressed algorithmically by always loading as many vertices as possible, deferring the frontier boundary check to the compute phase. Overall, the optimization of the tracker for the NVIDIA platform has a larger effect than the PREM overheads, and *Coal-PREM* is still 2× faster than the vanilla version of *Naive*.

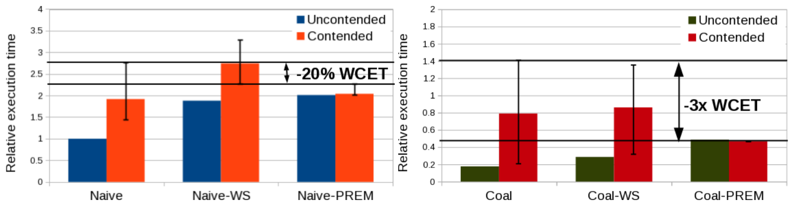


Figure 2.13: The effect on *Naive* (left) and *Coal* (right) under memory interference from the CPU.

Freedom from Interference

We now evaluate the effectiveness of PREM to achieve freedom from interference, as well as the performance benefits achieved under memory interference, due to stable execution times as opposed to accepting increased latencies. Each implementation is executed with and without memory interference, generated by `stress` [55] executing on the CPU. This is a system stress testing tool available on most GNU/Linux systems, including NVIDIA's Linux4Tegra, capable of generating large amounts of memory requests, cpu load, etc. For these experiments `stress` executes 24 threads, each accessing a 32 MB array in strides of 129 bytes.

The results are presented in Figure 2.13, in which all execution times are normalized to *Naive* without interference. For both *Naive* and *Coal*, the PREM version has significantly lower sensitivity to interference than the other versions, where the base versions suffer a slowdown of 2.7× and 8× respectively. In the case of *Naive*, the execution time of the PREM version is only 20% lower than the base version, because the execution time is already dominated by memory inefficiency and thus the difference is low. In contrast, *Coal* that has very efficient use of memory is very sensitive to memory interference, and the ability of the PREM-enabled version with *GPUguard* to keep the execution time stable also under interference makes for a large difference. In this case, the PREM version with *GPUguard* is 3× more performant than the base version, including the overheads discussed in the previous section.

The execution time of the PREM versions remain constant with and without interference – which is the expected result – for all except one measurement of *Naive-PREM*. The outlier is due to the interrupt latency variance in the default Linux4Tegra Linux kernel. This issue can be addressed with kernel extensions, such as *PREEMPT_RT*, which bound these latencies for real-time Linux systems.

Furthermore, the effects of memory interference to the warp specialized version is similar to that of the base version, showing that the code transformations in themselves do not limit the interference, but the support of *GPUguard* is required to achive these results.

2.3 Conclusion

The work in this chapter has demonstrated that CPU and GPU memory accesses can be executed in a timing predictable way, by enabling PREM-compatible exclusive access to the shared memory system during their *memory phases*. By dimensioning the GPUguard scheduling quantas to the WCET execution times of the phases in isolation (i.e., single-core equivalent timing analysis) near-zero variance in execution time is achieved also under memory interference.

We have seen that compute intensive tasks like matrix multiplication (Section 2.1.4) can be executed with GPUguard in a PREM-compliant way at almost no overheads. For more memory bound workloads, such as the path planner (Section 2.2.3) the shorter execution time of compute phases leads to worse amortization of the synchronization overhead, leading to larger overheads when condering execution in isolation, i.e., with no memory interference, the insights provided in Section 2.1.2. In both cases, however, it has been shown that PREM-like execution with GPUguard can provide significant improvements in the measured WCET under interference, thus achieving its set out goal.

The fundamental technique used to transform GPU kernels in this section has been warp specialization. This technique was initially proposed to overcome the lack of hardware DMA engines in NVIDIA GPUs, limiting previous techniques for efficient data transfers. Here, we have made limited use of this technique, as a means to achieve separate memory and compute phases for PREM execution. However, a main feature of CudaDMA [53] that proposed warp specialization was to enable double buffering to improve memory utilization by efficiently transfering data during computations.

Such techniques would potentially be feasible for compute-bound kernels also under PREM, in which the compute phase is long enough to enable two memory phases to be executed under its duration. However, the introduction of such schemas could significantly limit the time share of the memory available to other parts of the system, while the improvements for the GPU kernels are relatively small – since the kernels are compute bound, the memory phases account for a relatively small portion of the execution time. For this reason, such approaches were not explored here, but may be relevant to explore as part of future work – especially in systems that provide cheaper synchronization primitives to minimize the overheads of PREM phase changes.

As memory interference is a significant problem in any system with shared memory, the technique is also relevant for other types of heterogeneous systems. While techniques such as warp specialization are specific to GPUs, any other techniques can be used to transform programs into separate memory and compute phases. Thanks to the portable implementation of the GPUguard synchronization scheme, it could be used with any other software-programmable heterogeneous system with minimal changes.

Overall, using the techniques in this chapter PREM can be applied to heterogeneos systems in much the same way as previous work has enabled its application to multi-core systems. PREM by design removes memory interference effects between tasks in a system, providing single-core equivalence in timing analyses leading to system composability. Being able to apply single-core equivalent techniques to systems as complex as heterogeneous CPU+GPU architectures is a major benefit of the work presented in this chapter.

Having demonstrated the feasibility of the approach at runtime, the next chapter addresses the natural next step: In order to achieve the results in this section significant code refactoring was required. Wider adoption of the technique could be facilitated by compiler support to handle this tedious task.

Chapter 3

PREM Compiler Support

The transformation of legacy programs into PREM intervals, that preserve correctness of the original program and respect the sizes of local memories is a tedious and error-prone task. Already in the original proposal for PREM [24] it was expected that compiler support would eventually be required to make this technique applicable in practice. Before our work started, the only automatic code generation technique available for PREM was LightPREM by Mancuso et al [56]. This is a profiling-based technique that profiles memory accesses during runtime, and remaps run-dependent addresses to known fixed addresses in the program address space. From this profile, LightPREM injects prefetch phases into the binary of the program. This approach has a significant drawback in that prefetching of incorrect data found during profiling leads to segmentation faults, and any accesses that do not appear during every run of the program must be purged from the memory phases. This has two drawbacks, first that this leads to incomplete prefetching for PREM, potentially limiting the isolation of the interval partitioning, and second that the memory phases created from black-box observations may lead to unknown segfaults as the program input changes.

Compilers instead, have a full visibility of the code under compilation, and through static analysis techniques can generate memory phases that are correct under any execution. Constructing a PREMizing compiler is therefore a significant improvement over the state of the art in automatic PREM code generation from legacy software. To transform code into PREM-compliant versions, a PREM compiler requires means to analyze the code and divide it into PREM intervals, a method to automatically transform the code into *prefetch*, *compute*, and *writeback* phases, and a means to ensure that the PREM time-separation of memory phases can be upheld at runtime.

This section starts by presenting a GPU-centric PREM compiler, that divides GPU kernels into PREM intervals based on parallel loop constructs from high-level programming languages, e.g., OpenMP. This compiler immediately follows the observations from the previous chapter, automating the PREMization that was there performed manually. Targetting the same architecture, it uses GPUguard to enforce the PREM memory and compute phase separation.

Following this, the compiler techniques are extended to support more general programs and additional platforms, such as general purpose multi-core CPU and programmable many-core accelerators (PMCA). This section further provides insights on the architectural awareness required within the compiler to generate well-performing code for platforms with different compute and memory hierarchies.

The compiler techniques presented in this chapter are then evaluated in the following Chapter 4.

3.1 A Heterogeneous PREM Compiler

This section describes the work presented initially in *DATE'18* [57], and extended and published in the *IEEE Transactions on Computers* [40] in 2020, aimed at producing a PREM compiler for heterogeneous CPU+GPU systems. In this section, the focus is completely on the GPU compilation, and we will extend it to also encompass the CPU in Section 3.2. To simplify programmability, we build upon recent proposals for directive-based programming models [58, 59, 60] – which are more abstract than the low-level coding style of CUDA [43] or OpenCL [61] – 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 [58]. The programs generated by the compiler are compatible with *GPUguard*, as presented in the previous section.

3.1.1 Compiler Design Decisions

Following the discussion in Chapter 2, and taking into account the capabilities of compilers to analyze and transform code, we base the compiler design on the following pillars:

Real-time coding standards – To be able to transform legacy code to PREM-compliant code, the compiler must be able to infer many properties of the code. In the general case, this can be impossible to achieve, due to non-deterministic program flow and dependency on runtime data that is not available at compile time.

To limit the impact of such factors, we limit the scope of the PREM compiler to code that has been written in accordance to best-practices for real-time systems, such as the MISRA guidelines for the Automotive industry [62]. 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. This means that such code is more subjectible to exact analysis tools can be used in place of, e.g., profile-based techniques. Current compiler infrastructures such as GCC [63] and LLVM [64] contain built-in analyses to understand control flow, variable evolutions, etc. to determine such information.

Limiting the scope of the PREM compiler to code written accoding to real-time standards is reasonable, as the target of PREM is the enabling of real-time execution, thereby matching well with such real-time coding best practices.

High Level Language Compilation – In the previous chapter we applied PREM transformations directly to CUDA code. While it would be possible to design a heterogeneous PREM compiler on top of CUDA, this would negatively impact the programming model. Languages like CUDA, OpenCL, etc, offer a low-level programming paradigm, which gives the programmer significant opportunity for workload partitioning, and low-level optimization. As the PREM transformations are applied, such low-level optimizations may be found contrary to both performance and predictability in the transformed code. Furthermore, were the compiler to undo such explicit programmer decisions, there would be a mismatch between the code written and that of the final program. Such mismatches are in stark opposition to real-time best-practices for coding, which give a high value to code readability and understandability.

At the same time, the push towards the adoption of directive-based programming models such as OpenMP [58] and OpenACC [59] in the context of GPUs is constantly growing. 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]. By adapting such high-level languages as the basis for the PREM transformations, not only do we levereage this *per-se* valuable feature, 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 programmers. Already today the high-level language compilers are performing such decisions based on performance optimization, and thus is a good match to the needs of PREM transforming compilers.

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 Antao et al [65], 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. In the case of traditional discrete CPU+GPU systems, the offload boundary is the only point where CPU-GPU interference needs to be considered, as the discrete memory of the GPU does not give raise to interference during kernel execution. This is not the case in integrated CPU+GPU platforms where the DRAM is shared. As outlined in Section 1.1, all memory requests are served by the same physical memory, and interference occurs throughout kernel executions. In addition to

[1]Advanced directives provide detailed control to expert users.

[2]*num_threads* and *num_teams* (corresponding to CUDA *blockDim* and *gridDim*), and other OpenMP clauses that can be applied to parallel for loops.

this, the local memory λ of the GPU is not addressable from the host, in contrast to the GPU DRAM in discrete GPU systems. Therefore, data can not be copied into the scratchpad using the `memcpy` functionality. This means that in both techniques, the kernel code needs to be transformed not only to express intervals that are small enough to fit within the size of the local memory $size(\lambda)$ (Section 1.4), but also to manage the data transfers to local memory λ.

As GPU kernel code needs to be transformed in either technique, and the data movements need to be transformed from GPU side, it makes little sense to split kernels and manage PREM at an offload granularity. As has been argued, the contents of the kernel code would be the same in either case. Therefore the only difference between either controlling PREM phases from within the GPU kernels – i.e., using the GPUguard techniques – or at a per-offload level is the addition of further offloading overhead due to GPU reconfiguring at each interval boundary. Such operations are furthermore part of the closed-source driver, and can not be relied on for timing predictable execution, unless given by the manufacturer.

Following the selection of high-level OpenMP programming models, the PREM transformation into intervals can be easily achieved based on 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. This encodes the PREM interval synchronizations *continously* within the kernels, given by synchronization points explicitly expressed in the code. While kernels are reduced to a single offloading point, this point still requires protection to avoid interference due to OS scheduling jitter, i.e., interference from other tasks competing for processor time, but this problem is greatly reduced with less frequent offloads.

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 we will show in Chapter 5 that GPU caches could in theory be used for PREM, it puts additional constraint on kernels to ensure cache-friendly access patterns. For this reason, and as the main goal of the presented work is to achieve predictability,

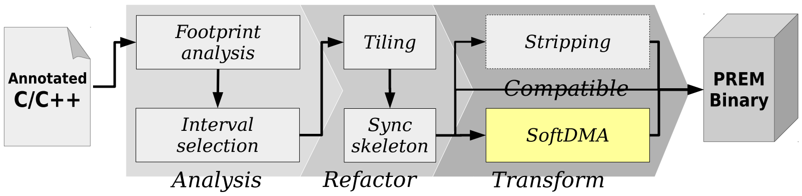


Figure 3.1: The analysis and transformation steps taken as part of the PREM-enabling compiler passes.

we have opted for the software managed SPM, that is not subject to un-controllable hardware eviction policies. This is consistent with the choice we made in Chapter 2.

3.1.2 Compiler Design

Based on these pillars, we identify the different phases that are required for a compiler to perform these code transformations. These phases can be divided into a three-phase compilation process, presented in Figure 3.1, which will be presented and motivatied in this section.

Before looking at the details, we outline the different phases of the PREM compiler, and motivate why they are required. The first phase is the analysis phase, which collects the necessary information about the program to perform legal PREM transformations. Since the size of an interval i, $size(i)$ must be smaller than the size of the local memory λ, $size(\lambda)$ (see Equation 1.1) the first step of the analysis phase is the calculation of the *memory footprint* of each code segment. As outlined in Section 3.1.1, the presented PREM compiler uses loop iterations as the atomic construct from which PREM intervals are created, and thus this heavilty relies on loop analysis techniques. We will extend this phase to handle more general control flows in Section 3.2. Based on this information, intervals i that are valid with respect to Equation 1.1 can be *selected*. Due to the use of loops as atomic units, the main technique used for interval selection is *loop tiling*, i.e., dividing a larger loop into several smaller loops, each constituting a PREM interval. It is the job of the interval selection step of the analysis phase to determine which *tiling factor* that results in PREM intervals for which Equation 1.1 hold.

Based on the interval selection done in the Analysis phase, the Refactoring stage transforms the code such that it conforms to the structure dictated by the selected PREM intervals. Practically, this involves applying the selected *tiling* scheme to the original code. Following this, hooks for the PREM scheduling runtime are inserted to divide the interval into *prefetch*, *compute*, and *writeback* phases.

Finally, the *transformation* stage is tasked with generating the memory phases for each individual interval, and injecting them following the scheduling hooks inserted in the refactoring phase. We will describe three different ways of achieving this transformation; *stripping*, *compatible intervals*, and *SoftDMA*.

Control Flow Graphs

To provide visual information throughout the presentation of each of these phases, we will use a running example based around the *control flow graph.* This allows us to show the incremental changes performed by the PREM compiler throughout the compilation process to a general representation of a loop structure.

The CFG G is created for each function F in the program, and is a directed graph $G_F = \{B, J\}$ that consists of the set of *basic blocks* B (nodes), and the set of branches (or jumps) J (edges). A basic block $b \in B$ is an ordered sequence of instructions $inst_0, inst_1, inst_2, \cdots$ of arbitrary length, ending with a terminator instruction. Terminator instructions include return statements, and importantly branch instructions, which each correspond to a jump $j \in J$, which targets any basic block $b_{target} \in B$. As G is a directed graph, each jump $j \in J$ therefore has a source b_s and a target b_t basic block in B, and we say b_s is the *predecessor* of b_t, and correspondingly that b_t is the *successor* of b_s. The return statement exits the scope of the function F, and as such provides a *sink* for the paths through the CFG G_F.

A well-formed loop consists of at least four basic blocks. The first is the *preheader*, which performs the initialization of variables. This node preceeds the cycle in G that represents the loop, and has only one successor, the *header* block. The header block is the first block that is part of the cycle, and it contains the instructions necessary to test

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

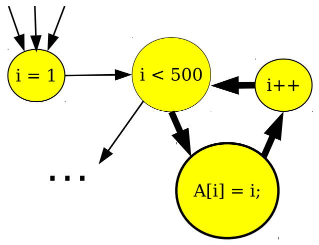


Figure 3.2: Original loop

the *loop condition*. The header block has two successors, depending on the outcome of the test. If the test result is false, the control flow jumps to the *exit node* of the loop, which is the first basic block following the loop that is not part of the graph cycle. If the test result is true, the control flow instead jumps into the cycle to the *loop body*. The loop body consists of one or more blocks and may have internally arbitrary control flow, including nested loop structures, as long as the final block in the *loop body* has an unconditional jump to the *loop latch*. The *latch* is the final necessary ingredient in a well-formed loop, and is responsible for *bumping* (e.g., increment, decrement, ...) the loop variables (as initialized in the *preheader* before unconditionally jumping back to the *header* (thus completing the cycle), and the *loop condition* is re-evaluated.

In loops in which the *loop condition* consists of evaluating an expression E of a variable V that is incremented or decremented by a constant value in each each cycle – loop iteration – we refer to variable V as the *induction variable*.

Consider Figure 3.2 that shows both `C` code and the CFG of a simple loop. In this representation, the loop is defined by an initial variable initalization `i = 1` in the *preheader* (here i is the *induction variable*), followed by the *loop condition* test `i < 500` in the *header*. The *body* of this particular loop consists of a single basic block which performs `A[i] = i`, before jumping to the *latch* which performs the bump of the *induction variable* `i++`. We will use this simple – but general – example throughout the following sections to illustrate the operation of the compiler.

3.1.3 Analysis and Refactoring

As outlined in Section 3.1.2, the first step in the analysis phase is to determine the memory footprint of the loops. The analysis of memory accesses in loops can be classified into one of two classes. The first is *loop invariant* accesses, in which each iteration of the loop accesses the same data. Examples of this include (but are not limited to) scalar accesses, or array accesses where the index does not depend on the loop iteration (e.g., constant, or not dependent on the *induction variable*). These accesses can be trivially identified from the internal representation of the compiler. The other class, correspondingly, are *loop variant* accesses, in which each loop iteration accesses a different datum. An example of this are array accesses where the index changes as a function of the induction variable, as for `A[i]` in Figure 3.2.

It is important to separate between these two classes to produce an exact footprint for the loop. If loop invariant accesses are accumulating the loop footprint for each iteration, the footprint will be overestimated, leading to small intervals and frequent scheduling decisions (adding overheads, as outlined previously in Section 2.1.4 and to be further explored in Section 5.1.1). Likewise, if the compiler does not account for all accesses that result from a single symbolic statement, e.g., i in `A[i]`, the footprint will be under-estimated, which may lead to violations of the PREM requirement expressed in Equation 1.1. This information can not be extracted directly from the internal representation in the compiler, but can be identified using built-in analyses, such as Scalar Evolution (SCEV) analysis.

Scalar Evolution

Scalar Evolution (SCEV) [52] analysis determines how the loop *induction variable (IV)* changes over the execution of the loop. Consider the example in Figure 3.2: Within the compiler, i is represented by an ambigous symbol, obscuring which elements of array A that will be accessed at runtime. SCEV analysis uses the available information in the *preheader* to determine that at the start of the loop, $i = 1$, together with the information expressed in the *latch* to that for each subsequent iteration $i_n = i_{n-1} + 1$, enables the obscure i symbol to be replaced with a rich *SCEV expression* that describes this evolution of i over the loop, on the form $\{start, op, stride\}$. In this case, i is represented as $i = \{1, +, 1\}$. In addition to understanding the value of i at any arbitrary iteration, it allows further information such as the iteration count to be calculated, by testing at which iteration of the loop that the *loop condition* $i < 500$ becomes false.

To extract this rich information, 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. In the case where e.g., the exact start or end values of i (1 and 500) can not be statically determined, the SCEV expression will be a symbolic expression, which limits the amount of information that can be extracted. In the following discussions, we assume that the SCEV expressions are always non-symbolic, i.e., the induction variable's start and end values, as well as its stride can be exactly determined at compile time. This is in line with typical requirements on real-time code (see Section 3.1.1) although we will further discuss how to lift this limitation of scope in Section 3.2. Note however that the presented methodology does not fundamentally depend on Scalar Evolution, but other loop analysis techniques, including profile-based techniques or bounds provided through programmer annotations could be used to provide the necessary information.

Thus, by calculating the evolution of i the compiler is able to determine all memory accesses that arise from a symbolic expression like `A[i]`.

SPM Buffers and Memory Footprint

Now that all accesses in the loop can be identified, the footprint of the loop can be analyzed by the compiler. However, one additional piece of information is required for the footprint analysis to return meaningful information. This issue arises from the design choice of using SPM for local storage. As SPMs are explicitly addressed, the staging of data in the SPM requires the creation of buffers. At which granularity data is laid out in these buffers thus impact the footprint of an interval.

For this work, we have decided to create *dense* SPM buffers for arrays, to reduce the address calculation overheads (see discussion in Chapter 5). This means that the relative distance between two elements in the original array and the SPM buffer are preserved, i.e., `A[n]` and `A[n+C]` in the original array is represented directly by `A_spm[m]` and `A_spm[m+C]` in the SPM. This implies that the absolute footprint depends on the *range* of elements accessed in the array, as space is allocated also for unused data in between.

However, applying this strategy directly to the address of the access would not enable multidimensional access patterns (e.g., matrices) to be managed without blowing up the resulting memory footprint. Therefore, the strategy is not applied directly to the address, but to each individual dimension used to index into the array. For example, a typical matrix access `A[i][j]` has two dimensions (one per `[]`) and would be represented by a two-dimensional buffer that is dense in dimensions i and j, but not in the range of addresses that these represent.

This focus on dense data structures is motivated by their prevalence in typical GPU applications, as they result in coalesced access patterns that are a prerequisite for good performance. Thus, this approach presents no practical limitation to typical GPU workloads, and solutions for other architectures will be discussed in Section 3.2.

Footprint Analysis

Having addressed the analysis of loop variant accesses, and the impact of SPM buffering on the resulting footprint of the program, the footprint can be calculated. Algorithm 1 shows how to calculate the range of elements M accessed within an analyzed loop. Lines 6-7 use SCEV to determine the elements accessed by loop-variant statements 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 element 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 accessed data M.

Sub-loops are represented as new loop structures with their own *preheader* etc. as the *body* of the *parent* loop. Note that this means that for each invocation of the inner loop, the loop variant accesses of the *parent* loop may be loop invariant in the sub-loop. This means that these accesses result in a smaller range of accessed elements, in

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
```

turn giving the property that the size of a sub-loop is at most the size of its parent loop.

PREM Interval Selection

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.

There are two main cases for turning a loop L into an interval. The first case is if the memory footprint of the entire loop is small enough to fit into the SPM λ_{SPM}, $size(L) \leq size(\lambda_{SPM})$. In this case, the data accessed during the loop can be loaded in its entirety into the local memory using the PREM *prefetch* phase, wherafter the loop is executed as-is in the PREM *compute* phase, although, in the case of SPM, with pointers updated to access the local memory instead of DRAM. At the completion of the loop execution, the data is written back to DRAM using the PREM *writeback* phase.

The other case, when $size(L) > size(\lambda_{SPM})$, the loop must be divided into n smaller chunks $L \rightarrow l_0, \cdots, l_{n-1}$, such that each chunk l_m fulfills $size(l_m) \leq size(\lambda_{SPM})$. This is achieved through tiling, and each tile represents an individual PREM interval. As outlined in Section 3.1.1, the selection of tiling granularity, i.e., the number of iterations of L to execute in each l_m, directly corresponds to the selection of PREM intervals.

OpenMP Work Distribution for Parallel For

OpenMP describes data parallelism through loop structures. During compiler code generation this loop must be transformed into GPU code that distributes work over threads and blocks. As mentioned in Section 3.1.1, the distribution over blocks is done using the *teams distribute* pragma. The distribution of work to threads is however done using classical OpenMP loop scheduling. By assigning iterations (i.e., values of the IV) of the outer loop to different threads, the workload can be efficiently executed in parallel. For the purposes of this presentation we refer to this outer loop as the SIMT loop, as it distributes work across warp threads executing in SIMT mode (SIMT = Single Instruction Multiple Threads).

This solution does however have an additional impact on tiling for OpenMP GPU kernels. For the purposes of the following discussion we will therefore separate the management of iterations of the SIMT Loop from iterations of other nested loops, which are executed in-order internally within each thread, as one would expect from classical single-threaded loops. For this purpose we refer to the iteration space of the SIMT Loop as the *blockDim*.

When tiling the *blockDim*, we constrain the tiling factor to be a 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.

Tiling

Using this information, Algorithm 2 shows how the memory footprint is used to achieve load balanced tiles, or PREM intervals. Lines 1-6 handle the case when the tiling is performed on the *blockDim*, for which we select the largest number of *block iterations* that produces a tile that fits in the SPM (line 3), i.e., fulfilling Equation 1.1. If no such tile can be created, the algorithm returns a *failure* (line 5). Note that

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: $\lambda_{SPM}$ is the size of the local scratchpad
 1: if Loop is SIMT Loop then
 2:     if $Footprint$(Block iteration of Loop) $< \lambda_{SPM}$ then
 3:         Set $blockDim$ to largest multiple of $num\_threads$ such that $Footprint(Tile_{blockDim}) < \lambda_{SPM}$
 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 $Footprint(Tile_{blockDim \times tileDim}) < \lambda_{SPM}$
11:     end for
12:     If $V < 1$ Then Failure EndIf
13: end if
```

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[3].

Lines 7-13 of Algorithm 2 handle the tiling of N levels of inner loops. To respect the SIMT loop we start by assigning a single *block iteration* to 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 $tileDim_n$ ($n = 1 \ldots N$), 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, λ_{SPM} (line 10). If it is not possible to find V such that the loop fits into λ_{SPM}, we use $V = 1$ if the loop has subloops, and recurse to the subloops on lines 9-11. This is motivated by the fact that for each recursion the footprint of an interation decreases, as outlined in Section 3.1.3. If there are no subloops (i.e., innermost loop) and we cannot tile it to be smaller than λ_{SPM}, setting $V = 0$ triggers a Failure on line 12[4]. Ultimately, the inequality in Equation 3.1 must hold, meaning that

[3]In practice, none of the kernels we will show in Chapter 4 trigger this case.

[4]Addressable by splitting the inner loop into two PREM intervals, using the techniques presented in Section 3.2.

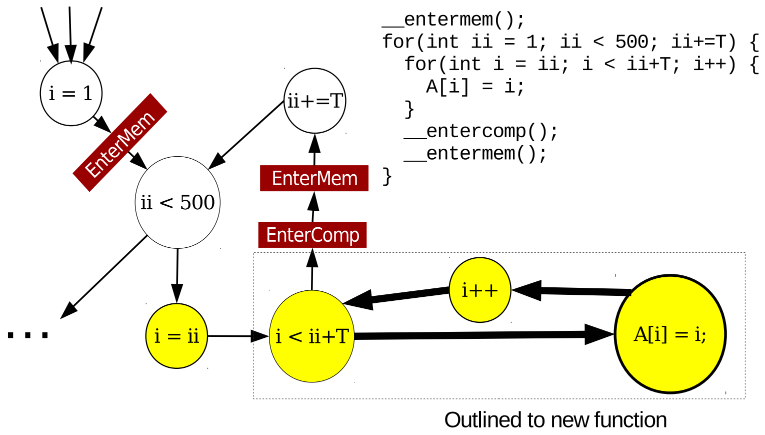
the more threads that are used per cluster (*blockDim*), the smaller the *tileDim* (iterations local to the thread) will become.

$$\lambda_{SPM} \geq blockDim \times tileDim = blockDim \times \prod_{n=1}^{N} tileDim_n \quad (3.1)$$

Tiling lives at the boundary of the analysis and refactoring phases of the PREM compilation, as outlined in Section 3.1.2. The decision at which granularity to perform the tiling – and thus create the PREM intervals – is taken as part of the analysis. The first step of the refactoring phase is to transform the code to match the tiling parameters derived during the analysis phase. Returning to our running example, as shown in Figure 3.3, the tiling of the loop adds the uncolored nodes in the CFG, and the inner loop in the C code. As part of the tiling process, the tiled loop l_n is *outlined* to a new function (this is the opposite of *inlining*), as shown by the dotted rectangle in Figure 3.3. Note that our approach is not specific of any tiling technique, and more advanced techniques [66, 67] can be used.

Synchronization skeleton and Scheduler Hooks

Following this, as the next step in the refactoring phase, the resulting tiles are separated through the insertions of synchronization calls according to the GPUguard protocol, as presented in Section 2.1. These synchronizations are concretely represented by function calls into a separate PREM library, which we will further describe in Section 3.2.4. Through these synchronization points, the GPU program is effectively divided into PREM memory and compute phases, as shown in Figure 3.3 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. Based on this synchronization skeleton, the following transformation phase is used to transform the code into separate PREM *prefetch*, *compute*, and *writeback* phases, as outlined in Section 1.3.



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

Figure 3.3: Tiled loop with *EnterMem*/*EnterComp* sync skeleton.

3.1.4 Transformation

The goal of the Transformation phase is to separate the created tiles – PREM intervals – into separate memory and compute phases. However, at this point it is worth noting that the insertion of the GPU-guard synchronization skeleton, performed already during the Refactoring phase, already is a valid representation of the PREM *compatible interval* as described in Section 1.3.2. While these interval do not benefit from the ability to continue operation on local memory while other tasks are using the main memory, it is perfectly possible to use GPUguard to isolate the memory accesses – i.e., the entire legacy-style execution of the kernel – and thus it fulfills the goals of isolating memory operations between tasks in the system that PREM requires.

Thus, to create a compatible intervals, no further transformation is required, and the code can be directly emitted, as illustrated in Figure 3.1. While the foreseen use of these compatible intervals is only to support legacy code that for some reason cannot be turned into predictable three-phase intervals, as outlined in Section 1.3.1, it is useful to be able to emit them from the compiler anyway to enable comparison, as we will show in Chapter 4. It is worth noting that the

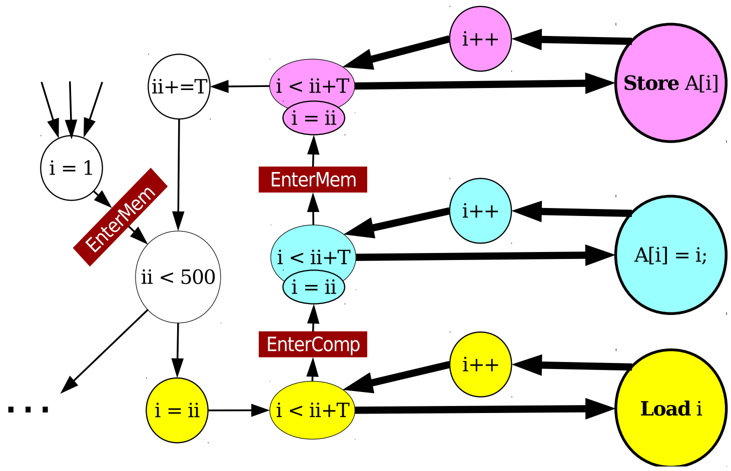


Figure 3.4: Skeleton extended to Load Execute Store phases, with *EnterComp* and *EnterMem* synchronizations.

tiling failures described as part of Algorithm 2 can be handled with compatible intervals, as these are not technically limited to the size of the local memory $size(\lambda_{SPM})$, as they operate directly on DRAM. Furthermore, in Section 3.2 we will further extend the compiler to apply this technique to handle such and other analysis phase failures.

A visual representation of how compatible intervals created this way are executed in the context of GPUguard synchronizations is presented in Figure 3.5a.

Three-Phase PREM Intervals

The main goal of the PREM compiler, compatible intervals aside, is to create three-pased intervals with full *prefetch*, *compute*, and *writeback* phases. This section presents two fundamental techniques to achieve this, as well as different ways to generate code to achieve different execution schemes at execution time. We refer to the first technique as *stripping*, and the second technique which improves the efficiency of the memory phases as *SoftDMA*.

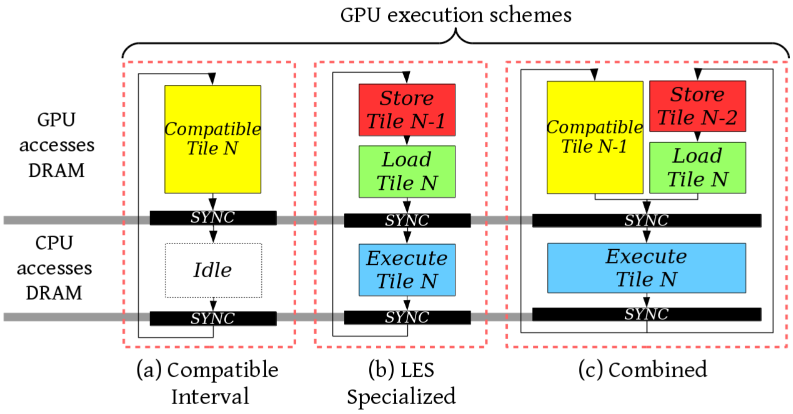


Figure 3.5: A logical view of different GPU execution schemes enabling mutually-exclusive access to the system DRAM with the CPU executing in the background.

Stripping-based Techniques The easiest way to achieve the phase separation is to clone the tiled loop into two additional copies, and placing one after each synchronization, as shown in Figure 3.4. The PREM memory phases can then be created using a class of techniques which we refer to as stripping, and represented by the Decoupled Access Execute technique [68]. This technique transforms the PREM memory phases, by transforming the cloned copies of the original tile, and *stripping* the non-load/store code out of the respective phases, staging all loaded/stored data through the SPM, and transforming the original tile (now the *compute* phase) to access the staged SPM data. While intuitively *predictable intervals* created in this manner imply code transformation overheads compared to compatible intervals, they reduce the idleness in the system through continued execution while memory access is not granted, as shown in Figure 3.5b.

Additionally, by using the warp specialization technique presented in Section 2.2.1, the compiler can further 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 three-phase tile created through stripping, staging data through the SPM then computing locally in the *compute phase*. This is shown in Figure 3.5c. This technique was created to address the ineffective use of the memory bandwidth due to the stripping technique: The technique had previously been used successfully on cache-based general-purpose CPU programs, but it failed to make effective use of the GPU memory bandwidth. The *combined* technique addresses this by using the slack bandwidth to compute one tile in DRAM while performing the memory operations of the three-phase predictable interval. The effects of this will be shown in Chapter 4.

SoftDMA The need for the *combined* schema in combination with the stripping technique illustrates how this originally successful CPU technique failed to make effective use of the memory bandwidth of the GPU. 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 *prefetch* and *writeback* 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 [53]. 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 Figure 3.6a. 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 Figure 3.6b. In contrast, *SoftDMA* enables each unique accesses to be mapped to a single thread, as shown in Figure 3.6c.

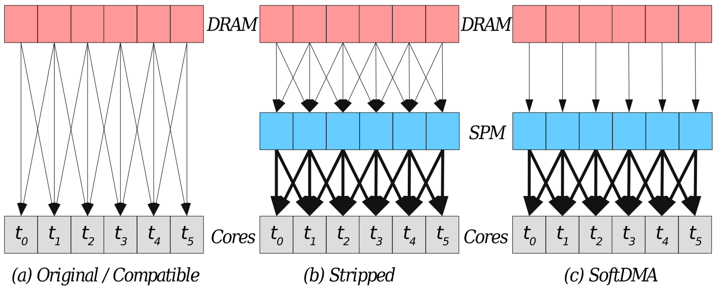

DRAM
DRAM
SPM
Cores
Cores
t_0 t_1 t_2 t_3 t_4 t_5
(a) Original / Compatible
(b) Stripped
(c) SoftDMA

Figure 3.6: The memory access patterns of the original program, and the PREM memory phases created with the *stripping* and *SoftDMA* techniques.

Instead of reusing the original control flow for the *prefetch* and *writeback* phases, as was the case for the *stripping* technique, optimized *SoftDMA* memory phases are created as loops in Algoritm 3. These loops mimick the behavior of DMA engines, and represent the smallest amount of code necessary to move the data. 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 += $step$), on the form outlined in Section 3.1.2. $L.iv$ is the IV (Section 3.1.3) 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, following the principle outlined in Section 3.1.3. 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). Note 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.

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. Instead, the *stripping* technique is used as fallback solution, if analysis provides insufficient information for SoftDMA code generation, as outlined in Section 3.1.3.

The use of *SoftDMA* only affects the Memory phases, i.e., the *prefetch* and *writeback* transformations. The only transformation done to the *compute* phase is to replace all accesses to global memory with accesses to scratchpad buffers, through which the memory phases stage data. The SoftDMA transformation does not rely on warp specialization. Visually, while the implementation is different, the exe-

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
```

cution scheme of SoftDMA is represented by the same scheme as the standard stripped representation shown in Figure 3.5. In this case, there is no need for a *combined* schedule, as the SoftDMA memory phases are already able to saturate the memory bandwidth, removing the need to execute a compatible interval in parallel. We will show that this is true in Section 4.1.3.

Having reached the point where the memory phases for each selected interval is created, the compiler proceeds to code generation in the backend, using the standard techniques already available. Together with the runtime libray to be presented in Section 3.2.4 the PREMized GPU code is ready to be scheduled and executed.

3.2 Extending the PREM Compiler to General Purpose CPU code

The techniques presented in the previous section enable the efficient PREMization of GPU kernels from high-level languages, as we will see in Chapter 4. From the analyis phase PREM intervals are identified, the refactoring stage restructures the code to match the granularity of the PREM intervals, in this case through the act of tiling, whereafter the transformation phase generates the necessary PREM memory phases and instruments the code to interact with the scheduling, e.g., GPUguard, infrastructure.

Before presenting the experimental results, in Chapter 4, this section shows how the compiler can be extended to PREMize not only loop-based GPU kernels, but also CPU code. This section is presenting work submitted to *ACM Transactions on Embedded Computing Systems (TECS)* [69]. The necessary phases of a PREM compiler, as shown in Figure 3.1, also applies for the PREMization of more general applications, but each individual phase needs to be extended. How this is done is presented in the following section.

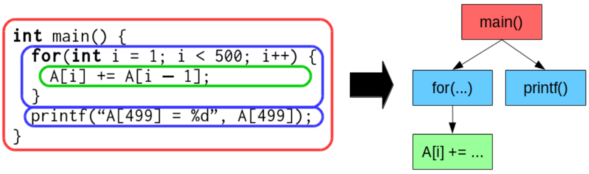
3.2.1 Design Decisions

To enable meaningful CPU PREM transformations, we extend the design decisions for the compiler from Section 3.1.1. Some design decisions remain valid also for CPU compilation, such as the reliance on real-time coding standards. This design decision remains, as the purpose of this is to enable static code analysis to enable an efficient analysis phase. Furthermore, we continue to focus on the same type of languages, e.g., C-based OpenMP programs. Fundamentally, the techniques presented in this section would be applicable to any compilation that lowers into LLVM IR, but the focus on C language family is consistent both with what is often used in practice, as well as the underlying concepts identified in the real-time coding best-practices. Furthermore, it has been shown that OpenMP is suitable for use in critical real-time systems [70].

Extended PREM Enforcement Granularity With respect to the PREM enforcement granularity however, focusing only on loop structures is no longer feasible, as CPU programs are not offloaded at such granularities. Instead, the entire program has to be taken into account. For this reason, the granularity of PREM intervals is extended from loop structures to the concept of Single Entry Single Exit (SESE) Regions (which will be further described in Section 3.2.2), in accordance with the related work by Soliman et al [32]. The use of SESE regions has two fundamental benefits. First, the SESE region structure is consistent with the coding standards for real-time systems [62], providing a framework in which the coding style and representations in static analysis tools and compilers are well-aligned. The second benefit is that SESE regions can be represented as a tree structure, in which one region (e.g., a function) encloses another region (e.g., a set of instructions). This hierarchy means that as the tree is traversed downwards from the root, the memory footprint of each region is smaller or equal to that of its parent – preserving the property that we previously applied to loop nests. Likewise, the previously developed loop analysis retains an important role in the extended compiler, as it remains applicable to the compiler analysis as soon as a SESE region that expresses a loop is encountered.

Support for Cache-based Memory Hierarchies Another important design decision is to extend both the footprint analysis and the transformation phase of the compiler to address cache-based memory hierarchies. The main reason for this is that CPU memory hierarchies are almost exclusively based around caches, as the use of SPMs require significant support from the OS to manage data locality [71]. Furthermore, the previously presented footprint analysis for SPM based systems (Section 3.1.3), optimized for efficient buffer creation for dense access patterns, is no longer adequate. As caches are implicitly addressed this design decision is not necessary for caches, as sparse access patterns do not imply an overhead in addressing instructions. Instead, cache line locality needs to be taken into account. For this reason, going forward we refer to the SPM-centric footprint analysis of the previous section as the *dense* footprint analysis, and the cache-centric footprint analysis to be presented in this section as the *sparse* footprint analysis.

Handling of non-Analyzable Code Regions with Compatible Intervals Lastly, as already identified in the original PREM proposal [24], an arbitrary general-purpose code can not be guaranteed to be fully statically analyzable. As we will outline later in this chapter, there are several reasons where such an assumption may break. For this reason, when purposing a PREM compiler for general-purpose



```
int main() {
  for(int i = 1; i < 500; i++) {
    A[i] += A[i - 1];
  }
  printf("A[499] = %d", A[499]);
}
```

Figure 3.7: A region tree and the corresponding source code.

code, it must be able to address such obstacles, and there are multiple ways to do it. For our compiler we have decided to make any code region which the compiler failed to analyze into a *compatible interval*, as presented in Section 1.3.2. This enables the PREM compiler to produce warnings for code which does not correspond to the necessary standards for static analysis [62], while still producing runnable binaries for any program. Of course, code regions that are turned into compatible intervals do not benefit from continued execution when main memory access is blocked, but in our opinion this is preferable to failing the compilation. Based on the produced warnings a system designer can determine if the application of compatible regions is adequate, or do the necessary changes to enable predictable intervals.

3.2.2 Extended Analysis Phase

This section presents the extensions to the analysis phase that are required to extend the compiler support from loop structures to generic programs. It presents how region trees provide the structural information for interval selection, how the footprint analysis is updated to account for cache-specifics, as well as a novel interval selection algorithm that operates on this information.

Program Representation

This section describes how we extend the fundamental representation of the program from being limited to loop structures to incorporate all parts of the program. The fundamental concept that we extend are Single Entry Single Exit regions.

Single Entry Single Exit Regions SESE regions are code regions represented by every part of the Control Flow Graph (CFG) that only has a single incoming edge and a single outgoing edge [32]. The SESE property maps well to PREM intervals, as data can be loaded on the incoming edge and stored on the outgoing edge. Regions are created by identifying the largest possible *non-overlapping* subgraphs $G'_0, G'_1, \cdots, G'_n$ in the CFG G_f of function f which only has one incoming edge J_{entry} and one outgoing edge J_{exit} to/from basic blocks not in G'. Non-overlapping means that a basic block $b \in B$ in CFG $G = \{B, J\}$ only appears in at most one of the sub-graphs G'_m. This process can then be repeated recursively by finding the largest non-overlaping subgraphs $G''_0, G''_1, \cdots, G''_n$ within each region graph G'. This creates a tree structure Υ_f for each function f where each level of the tree represents the sub-regions G' of the parent region G, as shown in Figure 3.7. The smallest unit that can describe a SESE region is a single basic block $b_{leaf} \in B$ that fulfills the SESE property, i.e., in the CFG $G = \{B, J\}$, there is a single edge $j_{entry} \in J$ that has b_{leaf} as target, and a single edge $j_{exit} \in J$ that has b_{leaf} as source.

SESE region analysis is built into modern compilers [72], but operate only at a per-function level. To construct a tree of the entire program, and in particular include any function calls made within a task τ, we use the technique proposed by Soliman et al [32] to nest the region trees of individual functions in accordance with the program Call Graph. For each basic block $b_{callpoint}$ that contains a call, the block is split at each *call point* (where a function is called) into three new blocks b_{pre}, b_{call}, and b_{post}, such that b_{call} only contains the call. Any preceeding instructions are moved to b_{pre} and any postceding instructions are moved to b_{post} blocks. The new blocks are linked with unconditional jumps, i.e., new edges in J. From this it follows that b_{call} is a SESE region (with the edge from b_{pre} as single entry and the edge to b_{post} as single exit), making the b_{call} block a leaf region of the region tree Υ. To construct an interprocedural region tree, the region tree $\Upsilon_{f_{called}}$ of the called function f_{called} is appended as a subtree of the region tree of the calling function f_{callee}, replacing the leaf region b_{call}, creating a *nested region tree.*

We further preprocess the code to accomodate special handling of branches. Each basic block $b_{condbranch}$ that terminates with a conditional branch j (i.e., corresponding to a branch in the control flow graph) is split into two basic blocks b_{pre} and b_{branch}, where b_{pre} contains all instructions from $b_{condbranch}$ leading up to but not including j, and an unconditional branch to b_{branch}. The new b_{branch} block only contains the conditional branch j. Thus, the operation is preserved, but it ensures that the region G (starting at b_{branch}) that contains the branching outcomes $G'_0, G'_1, \cdots, G'_n$ does only contain the conditional execution paths – as the SESE region G ends when the branching paths reconverge.

Incorporating the analysis stage from the GPU compiler, which was limited to loops is straight forward. Well formed loops, as described in Section 3.1.2 also correspond to regions, as shown in Figure 3.7. The loop has a single entry to the *header* from the *preheader*, and a single exit from the *exit block*. Furthermore, the loop body itself is a SESE region, as it has a single incoming edge from the *header* and a single exiting edge to the *latch*. This means that the region tree for a loop structure is able to express the same information as the loop nest information used in Section 3.1, and the loop analysis remains applicable to loop regions. The region representation is however richer, and allows sub-regions to be described within the body of a loop, meaning that loop iterations no longer need to be considered as the atomic unit from which PREM intervals are created (as outlined in Section 3.1.1). Instead, the region analysis further enables PREM intervals to be created within loops, if they consist of multiple sub-regions.

We refer to the set of all regions in task τ as Υ_τ. This corresponds to the region tree Υ_{main} of the *entry point function* f_{main} of τ, i.e., the function that starts executing when control is passed to the task τ. This corresponds to the *main* function in a single-task binary, but may be any function inside the program that expresses multiple tasks. By means of *nested region trees*, this tree contains all regions of the task in the and enables interprocedural interval selection.

Interval Types and A Richer Representation In addition to the interprocedural nesting of the region tree, we further annotate the region tree with properties and connections to hold information useful for interval selection, which is to follow in Section 3.2.2. Figure 3.8 shows an example region tree with these properties and connections. First, we explicitly annotate the child region G' which is the entry point of a region G. There are two cases, depending on if b_{entry} has a single or multiple successors. In the case of a single successor, the entry point is the first basic block b_{entry} that is executed within G, and as regions are non-overlapping b_{entry} belongs to exactly one subregion G'. As an example, if a region G can be divided into subregions G'_0 and G'_1, both G'_0 and G'_1 are children, but the one that contains the first basic block b_{entry} in the CFG of G is additionally labeled as *entry*. If on the other hand b_{entry} terminates with a conditional branch, we know from before that the entry block b_{entry} only contains the branch instruction, and we instead label every successor basic block b as *entry*. This represents the fact that the outcomes of a branch are mutually exclusive execution paths.

Secondly, as subregions G' have an inherent order in $G = \{B, J\}$, due to the edges $j \in J$, we express this information on top of the tree structure with the *next* property. The property $AnextB$ is defined as the single exit edge of A being the single entry edge of B. The *next* property is only defined on subgraphs that have the same parent. This extension enables traversal of Υ both hierarchically (by leveraging the tree property) and in control flow order. This is highlighted in Figure 3.8 with the *next* arrows.

Lastly, we annotate the regions with their type, which will determine how PREM intervals can be selected from the region tree Υ. As outlined previously, *loop* regions will be handled specially by invoking the loop analysis presented in Section 3.1, and loops are therefore explictly labeled. Another construct that requires special handling are branches, as the branch outcome determines which of the child regions that are executed. Notice in Figure 3.8 that through the previous annotations we see the exclusivity expressed as the *branch* region having two *entry*-type children with no *next* relation. All other regions are labeled as *linear*.

Predictability Levels and Non-SESE Functions There are two main limitations with the interprocedural region tree that must be addressed to express complex and real applications. The first case is for regions with multiple returns f_{mulret}, as such functions do not

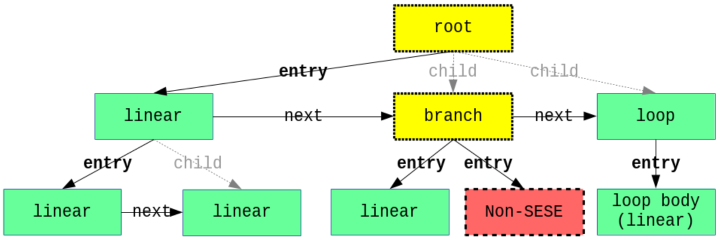


Figure 3.8: A region tree with several types and relationships.

fulfil the SESE property[5], and as such no region tree $\Upsilon_{f_{mulret}}$ can be derived and attached to the call point in b_{call} in the nested region tree. The second case is for calls to functions that are outside the scope of the compiler's *translation unit* (i.e., the file currently being compiled), which we will refer to as $f_{external}$. A clear example for this is in calls to functions in external libraries, e.g. syscalls. As the contents of the function are not visible to the compiler, it likewise cannot create a region tree $\Upsilon_{f_{external}}$ to attach to b_{call}. Here we discuss a conservative approach to ensure correctness also in the scope of such code, which can be commonly found in practice, and we will discuss more detailed solutions in Section 3.3.

To represent non-SESE code the region tree is extended with *non-SESE blobs* (marked red in Figure 3.8). These serve as placeholders, and will serve as hints to the interval selection that this code cannot be analyzed (non-SESE) or transformed (non-SESE and external). While this represents cases where the program cannot be transformed, there is nothing that can be done within the compiler to lift this limitation. For this reason, the compiler generates warnings when this occurs, such that the programmer can adjust their code to enable further analysis. Note that the underlying constructs that cause these limitations both are adviced against in real-time programming guidelines, and in line with the definitions of what presents a mandatory *compatible interval*, as outlined in Section 1.3.2 and described in the original

[5]Built-in compiler passes are available to transform multiple returns into a single one, limiting the impact of this factor in practice. However, in cases where such transformation fails, these cases must be handled.

PREM proposal [24].

To steer the interval selection process, following in Section 3.2.2, we introduce the concept of *predictability levels* to mark all *non-SESE blobs* as *Unpredictable*, all their parents and grandparents as *Mixed* (yellow in Figure 3.8), and the remaining analyzable regions as *Predictable* (green in Figure 3.8). This information is used in the selection process that we will present in Section 3.2.2.

This extended region tree provides the foundation for the PREM interval selection. However, before PREM intervals can be selected, the footprint of each node needs to be calculated. Already now we can see from the hierarchical structure of the region tree that it preserves the property we used in the loop based PREM compiler presented in Section 3.1, i.e., that traversing the tree downwards leads to region sizes decreasing.

Footprint Analysis

Memory footprint analysis is the process of calculating the size of the data accessed by each region in the tree, as required for PREM interval selection. How the memory footprint is calculated depends on if SPM or caches are used as local memory, through which data is staged.

Fundamentally, the software-management of SPMs means that data can be moved at any granularity, while caches operate at cache line granularity. Since accessing one datum will automatically bring in the entire cache line to the cache, the full size of the cache line needs to be accounted for, and to avoid returning overly pessimistic footprints, this means that the footprint analysis needs to account for spatial locality within cache lines. SPMs, on the other hand, are completely software managed, meaning that individual datum can be moved freely. On the other hand, as SPMs are explicitly addressed, well-allocated buffers need to be created to minimize addressing overhead, leading to constraints that must be accounted for in the footprint analysis. For example, it is possible to create sparse buffers in the scratchpad, but if this leads to significant addressing overhead (i.e., every DRAM address requires an expensive translation to the SPM counterpart), it may be better to allocate buffers with "gaps" in them, leading to an increase in the footprint compared to the size of the individual bytes accessed[6].

This means that a PREM compiler requires specific footprint analysis routines depending on which type of memory that is used on the system that code is being compiled for, and accounts for the *sparse* and *dense* footprint analyses as outlined previously. How this is achieved on the presented region tree is presented in this section.

Data collection Loop regions are analyzed using the loop based footprint analysis presented in Section 3.1. These *loop-variant* accesses are described by (*start*, *stride*, *tripcount*) tuples, which record the *source*-relative *start*, the *stride*, and the *tripcount* of each access pattern, where the *tripcount* describes the number of iterations of the loop. All identified memory accesses are stored in a memory access map $A = (start, stride, tripcount, size)$, corresponding to the information returned by the SCEV analysis for loop-variant accesses.

For scalar accesses we set $start = 0$, and $stride = tripcount = 1$, signifying a single access with an offset of zero from the *source* pointer. The *stride* will never be taken as the *tripcount* is one, but it is necessary for the footprint calculation below. For the calculation, the access map A also records the per-element *size* of each *source* data structure. For duplicate accesses to the same data element, the access map A only records the access once. Due to real-time coding guidelines, e.g., MISRA [62], it is often possible to follow the access back to its original *source*. However, if this is not possible, the compiler can not determine if two memory accesses are pointing to the same data structure, i.e., *pointer aliasing*. In these cases the compiler will conservatively over-estimate the memory footprint, by assuming that different data will be accessed.

From the access map A the memory footprint FP_R of each *leaf* region R is calculated. If SPMs are used, the footprint is calculated by Equation 3.2, summing over the range of addresses accessed in each data structure using the start s, stride t, tripcount c, and size b, in accordance with the *dense* analysis presented in Section 3.1:

$$FP_R = \sum_{S \in A_R} (\max(s + t \times c) - \min(s)) \times b \tag{3.2}$$

[6]Similar concerns are the fundamental reason why caches operate at a larger granularity than individually addressable bytes.

This represents the footprint for a single unified SPM buffer for each *source*, leading to reduced addressing complexity at runtime as there exists a 1-to-1 mapping for each data structure in DRAM and in the SPM. The use of a unified buffer may lead to an over-estimation of the footprint due to sparsity in the memory access pattern. As our focus for SPM is on GPU-style accelerators, this does not cause significant issues, as it is in line with how optimized memory accesses have to be constructed anyway, i.e., coalescing accesses as described in Section 2.2.2.

Extending the compiler with the *sparse* analysis for caches, where data movement is transparently handled by hardware without buffer address calculation overhead, no 1-to-1 buffer mapping is needed. Instead, the cache line locality needs to be taken into account, as data is moved at the granularity of cache lines – i.e., fetching a single element of four bytes will automatically bring a full cache line, typically in the order of 64-128 bytes, into the cache. This leads to the footprint calculation formula presented in Equation 3.3. The parameters are the stride t, tripcount c, and size b of each access in the access map A_R of region R, and the cache line size C.

$$FP_R = \sum_{(c,t,b)\in A_R} \begin{cases} (\lceil c \times \frac{t\times b}{C} \rceil + 1) \times C & \text{if } t \times b < C \\ c \times C & \text{otherwise} \end{cases} \tag{3.3}$$

If the cache line size and stride ratio is not taken into account, the footprint analysis might severely under-estimate the footprint of a region. Thus, for correct generation of PREM intervals, it is necessary for the compiler to be aware of the cache line size C of the target system. Furthermore, we conservatively assume unaligned memory accesses, by counting one extra cache line. Thus conformance to the PREM model is ensured, as the actual memory footprint is never larger than what is reported by the footprint analysis. Note that FP_R does not consider the set-associativity or replacement policies of caches, as these do not affect the memory footprint *per-se*, but only when data is evicted. This will be further explored when mapping the tasks to the real systems in Chapters 4 and 5.

For both caches and SPM, conformance to the PREM model is ensured, as the actual memory footprint is never larger than what is reported by the footprint analysis.

Footprint propagation As outlined in Section 3.2.2, the child nodes of a region R cover every basic block that is present in R. Thus, the footprint for the rest of the tree can be calculated by propagating the footprint of the leaf nodes upwards towards the root. For each region visited, the footprint contributions from all child nodes are merged, and all duplicate accesses pruned. This gives an important property of the region tree: *Traversing the tree downwards from the root means that the memory footprint of the regions are decreasing.*

Non- or partial loop analysis Static analysis methods, such as SCEV, are likely to fail for programs that have not been carefully engineered. To compile programs where some code footprint can not be analyzed, we extend *unpredictable regions* (Section 3.2.2) to include *footprint analysis failure.* any region for which the footprint analysis fails is demoted to an *unpredictable* region. During *footprint propagation*, the footprint is not propagated for nodes with one or more *unpredictable* children, as the contribution from the *unpredictable* region is not known.

Interval Selection

Once the region tree is annotated with memory footprints, the PREM intervals can be selected. This is done greedily by recursion on the region tree, as shown in Algorithm 4 (page 90), SIFR (Select Intervals From Regions), which returns a set of PREM intervals. This algorithm uses three helper algorithms. First, CRTI, Commit Region To Interval, which adds the given region to the current interval if the predictability level is the same. If the predictability level of the given region is different from the current interval, it creates a new interval with the correct predictability level, and adds the region to that interval. Second, MIS, Merge Interval Sets, which merges an interval set returned by a recursive call to SIFR with the current instance. Third, the function TILE which tiles a loop region to a specific size. Tiling is also known as *blocking* [52], and is typically used to increase cache locality. We use the same tiling technique as presented in Section 3.1, although without accounting for the *blockDim* if the loop is not parallelized using OpenMP.

The SIFR algorithm starts recursion on the root of the region tree. For each recursive call, it checks if *currentRegion* is suitable for a PREM interval. If it is, the region is selected and the recursion is stopped. If the region is not suitable, recursion continues to the children. Thus, the algorithm ensures that the entire program is covered in exactly one PREM interval.

If a region is suitable for a PREM interval depends on its predictability level. Most of SIFR handles *predictable* regions, starting on line 2. For *predictable* regions, SIFR first checks if the *currentRegion* is small enough to fit into the current interval, in which case it is added (line 4). If this is not the case, it checks whether it can start a new interval (line 6). If the *predictable* region is too large to fit into any interval, different steps are taken depending on the interval type. Loops are handled on line 10, where they are tiled to fit into intervals. At this point, the full loop is still represented by a single interval, but is expanded to encompass every tile iteration during the transformation phase. Note that *predictable* loop regions that are small enough to fit into a single interval are not tiled, but handled on lines 4 and 6. If a *branch* region (from control flow *fork*, to the *join*) is too large, the control flow of each branch outcome is separated into distinct chains of PREM intervals (line 14), to avoid prefetching data that is not on the current branch path.

For *linear predictable* regions too large for an interval, the recursion continues to its children on line 20. Note that parent nodes are larger or equal than all its children, and selecting all children (or grandchildren) implicitly selects the node itself. For leaf nodes, children are created by splitting the region as shown on line 23. As a leaf SESE region is always a single basic block, it retains the SESE property when split.

On line 30, regions marked as *unpredictable* are handled. At this point, there is nothing the compiler can do but to add them to a *compatible interval*. For intervals with *mixed* predictability, handled on line 32, we start by recursing to the child intervals, as we cannot create a *predictable* interval from a region which contains *unpredictable* code. During the recursion, the exploration of the tree will continue until it has reached a purely *predictable* or *unpredictable* sub-tree, at which point the previous steps can be applied. Once the recursive call returns, the selected intervals can be added to the set of intervals,

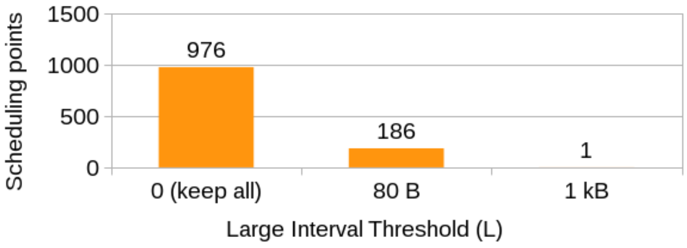


Figure 3.9: Reduction in scheduling points, through interval demotion for intervals smaller than L, for the task SENSOR.

as seen on line 35. Note that if the mixed predictability region is a *branch* region (as pictured in Figure 3.8), the same steps as for predictable intervals (line 14) must be taken, but they are left out to conserve space. This solution is sufficient to produce correct PREM code from *mixed* predictability regions. However, during our work with real applications we have determined that optimizing this case is paramount for performance. We will therefore revisit this step, covering lines 34 and 37 in the following section.

To ensure that the entire program is covered, on line 40, recursion continues, not on the *children* of the region, but on the successor (using the *next* relationship introduced in Section 3.2.2). This ensures that once the *currentRegion* has been successfully selected, interval selection continues in control flow order to the next regions. When Algorithm 4 returns, every region is assigned to a unique interval.

Interval Demotion

Returning to the selection of PREM intervals from *mixed* predictability regions, let us consider a fundamental trade-off of automatic PREM code generation. On one hand, *predictable intervals* are prefered, as these do not require mutual exclusion to the memory for their entire execution, enabling concurrent scheduling of other tasks. On the other hand, each interval requires a scheduling decision and a context switch, giving a strong preference for large intervals. From practical experience, the most common source of *unpredictable* intervals is

Algorithm 4 Select PREM intervals from regions.

```
 1: function SIFR(currentRegion)
 2:     if currentRegion.predLevel ← Predictable then
 3:         if currentRegion.footprint < availableMem then
 4:             CRTI (currentRegion, predictable)
 5:         else if currentRegion.footprint < totalMem then
 6:             start new interval
 7:             CRTI (currentRegion, predictable)
 8:         else
 9:             if currentRegion.isLoop then
10:                 TILE (currentRegion, totalMem)
11:                 start new interval
12:                 CRTI (currentRegion, predictable)
13:             else if currentRegion.isBranch then
14:                 for all entry ∈ currentRegion.entry do
15:                     start new interval
16:                     childIntervals ← SIFR(entry)
17:                     MIS (childIntervals)
18:                 end for
19:             else if currentRegion.hasChildren then
20:                 childIntervals ← SIFR(currentRegion.entry)
21:                 MIS (childIntervals)
22:             else
23:                 (left, right) ← SPLIT(currentRegion)
24:                 leftIntervals ← SIFR(left)
25:                 MIS (leftIntervals)
26:                 rightIntervals ← SIFR(right)
27:                 MIS (rightIntervals)
28:             end if
29:         end if
30:     else if currentRegion.predLevel = Unpredictable then
31:         CRTI (currentRegion, compatible)
32:     else if currentRegion.predLevel = Mixed then
33:         childIntervals ← SIFR(currentRegion.entry)
34:         if HASLARGEPREDICTABLE(childIntervals) then
35:             MIS (childIntervals)
36:         else
37:             CRTI (currentRegion, compatible)
38:         end if
39:     end if
40:     succIntervals ← SIFR(currentRegion.next)
41:     MIS (succIntervals)
42:     return committed intervals
43: end function
```

through syscalls to handle I/O, which will be a critical point for any embedded system. By definition [24], I/O operations are always compatible intervals, and thus *unpredictable* regions, but the calculations surrounding them are visible and analyzable by the PREMizing compiler, and will thus be represented as *predictable* regions to SIFR. In the context of I/O-heavy tasks, this causes fine-grained interleaving of *predictable* and *unpredictable* regions. If the compiler too strongly favors *predictable* intervals, it will lead to an explosion of small intervals, and the corresponding amount of scheduling decisions required by the runtime.

For this reason, the HASLARGEPREDICTABLE condition is introduced in SIFR on line 34. This function goes through the set of intervals returned from the children of the *mixed* region, and checks if the *predictable* intervals selected are larger than some configurable threshold L. If this is not the case, all child intervals are dropped, and the entire *mixed* predictability region is selected as a *compatible* interval (on line 37). We refer to this process of merging small interleaved *predictable* regions into a larger *compatible* interval as *interval demotion*.

Consider this illustrative example: We used our compiler to PREMize software components of an autonomous drone. One task of particular interest, SENSOR, preprocessed sensor input from gyroscope, accelerometer, and other sensors, before being used by the system. This task consisted of several thousand lines of code, had a requirement to execute with a period of $5ms$, and made heavy use of syscalls to perform I/O. Without *interval demotion*, the compiler would create almost a thousand intervals, as shown in Figure 3.9, as it would use every opportunity to create a *predictable* interval. Through *interval demotion*, it was possible to reduce the number of scheduling points significantly. Without *interval demotion* it would have been impossible to reach the $5ms$ period of the SENSOR task, as a scheduling decision was measured to take approximately $7\mu s$ on the target system – at 976 intervals the scheduling alone would have amounted to almost $7ms$. Thus, the trade-off between the benefits of *predictable* intervals and fewer scheduling points is crucial to consider for high performance.

3.2.3 Extended Transformation Phase

Extending the transformation phase for use on cache-based architectures is straight-forward: The PREM phase generation is simplified by the implicit addressing of caches. For SPMs it is neccessary to allocate buffers, filling them in the prefetch phase, refactoring the compute phase to use the data in the buffers, and write back the data in the writeback phase. For caches on the other hand no buffers need to be created, and the prefetch phase is created by simply adding a prefetch instruction for each data element that would previously have been loaded from DRAM and stored into the SPM buffer. Writeback phases can be created similarly using eviction instructions, if this is required on the platform as we will discuss in Chapter 4. Due to the implicit addressing, the compute phase does not require any instrumentation of loads and stores.

The fact that intervals are now expressed in regions instead of in loop regions does not affect how the synchronization skeleton is inserted, and the changes to the phase creation are minimal. First, the infrastructure developed for loops remains applicable also for loop regions, enabling tiling and phase generation in accordance with the techniques set out in Section 3.1. For non-loop regions, the SESE property provides natural insertion points for the memory phases at the single-entry and single-exit.

The stripping technique, adapted to emit prefetch/evict instructions instead of SPM load/stores, can be used to generate PREM intervals for cache based architectures. This approach is equivalent to the approach described in [68]. However, the previously presented SoftDMA technique needs to be updated to efficiently work with multi-core CPU tasks on cache-based systems.

SoftDMA on the CPU The primary motivation for SoftDMA when applied to GPUs was to ensure that all threads are fetching a unique piece of data at all times. However, the use of SoftDMA also has another important side-effect; it reduces the number of instructions that need to be executed.

This property is also required on the general-purpose processors that have been explored in this work, because the execution of a large amount of prefetch instructions can give multiple orders of magnitude

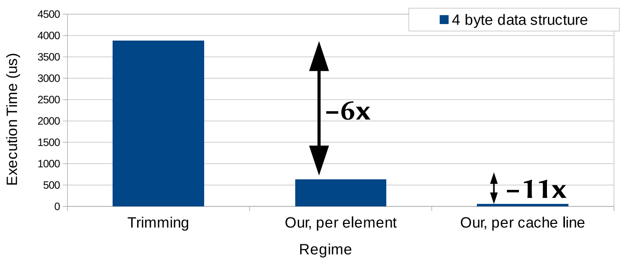


Figure 3.10: The execution time of the prefetch phase under three different *Prefetch phase* regimes.

hit on performance. The key thing to optimize this is to understand when data that is going to be prefetched is already in the cache, due to spatial locality of data that was already prefetched.

To address this, we extend the SoftDMA infrastructure presented in Section 3.1.4 to generate efficient memory phases also for non-GPU systems. To achieve this, we use the cache line locality information to generate minimal loops that during prefetching only touch each cache line onece. Cache line locality can be calculated within composite types like structs and arrays, but not across data structures or scalar variables. However, as composite data structures typically dominate the memory footprint, and they are unlikely to have cache line locality between them, this transformation still leads to large performance benefits for the PREM memory phases.

To illustrate the effects of this, we executed the 2DConv kernel from PolyBench [73], which has stencil-type accesses, and measured the execution time of the prefetch phases under three different regimes: *stripping* (as presented in Section 3.1.4), SoftDMA loops without cache line optimizations, and SoftDMA loops with cache line optimizations. The execution was performed on the ARM CPU on an NVIDIA TX1, and the results are presented in Figure 3.10. As can be seen, for this kernel, the transformation presented in this approach is 6× faster than *stripping* when not considering that multiple elements may be part of the same *cache line*. On top of this, the *cache line* optimization further increases the performance of the *Prefetch phase* 11×.

We will explain this effect in further detail, beginning with the $6\times$ improvement compared to stripping. The kernel considered is a 3×3 convolution kernel, meaning that during the execution of the kernel, each element will be accessed 9 times (disregarding elements on the border). Once as the center element, and eight times for the neighbor direction (north-west, north, north-east, east, . . .). Therefore, reusing the original control flow to prefetch this data means that each element is fetched 8 times more than needed. With our approach, we identify the exact memory access pattern, and can produce a *Prefetch phase* that only visits each element once. Thus, the upper bound on the improvement from this transformation is $8\times$, due to the removal of duplicate accesses, and the measured improvement is $6\times$.

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

HardDMA It is worth noting that the operation of SoftDMA is to emulate the operations of a hardware DMA engine, by iterating over the data structures to be copied. On platforms where such engines are available, the SoftDMA implementation can be easily adapted to perform 1D DMA transfers by instructing the DMA to perform these operations. As outlined in Section 3.1.4, SoftDMA creates N levels of loops for every N-dimensional data structure, which iterates over the data in each dimension. The inner-most loop will transfer data that is stored sequentially in memory, and as such this loop can be trivially replaced with a 1D DMA call. This enables PREM memory phase creation with hardware acceleration.

In addition to the trivial 1D DMA extension, support for DMAs that support multiple dimensional data structures has been explored by Cyrill Burgener [74].

3.2.4 Enabling Scheduling

In addition to performing the transformations, at the end of the Analysis phase a directed dependency graph is implicitly created from the PREM intervals that have been selected. Due to the selection process, each PREM interval, except for the program entry and exit, has predecessor and successor intervals implicitly defined through the control flow graph of the task. To schedule these PREM intervals, the scheduler requires a directed ayclic graph (DAG), however, the implicitly generated graph would contain cycles if loops have been tiled into several PREM intervals. For this reason, before the dependency graph is passed to the scheduler, these loops are unrolled, at a tile basis, to remove the cycles. Note that, even if the loops would not be fully unrolled within the code, for code size reasons, the dependency graph is always fully unrolled. Also note that, as loops that could not be analyzed are represented as a single compatible interval, these intervals do not need unrolling, as the DAG only represents the program at a PREM interval level. This process is required for the scheduler to work correctly, and is required even if the amount of unrolled intervals is very large. However, compared to loop unrolling, the unrolling on a tile basis produces much fewer nodes in the graph, and does not include instruction level information, but only interval identification information. Through this process, the possibly cyclic graph has been turned into a non-cyclic dependency graph, which is forwarded to the scheduler. Finally, each interval is assigned a unique id, which is annotated in the DAG, and also in the scheduling hooks inserted by the compiler – allowing the offline generated schedule to map to intervals executed at runtime.

PREM Scheduler Hooks

To facilitate the scheduling at runtime, the PREM transoformation has inserted the PREM scheduling hooks, as described previously. These are represented by function calls into an external statically compiled library, which we call *libprem*. This library consists fundamentally of a single function `__prem_notify(id, type)`. This function is responsible for implementing the runtime aspects of the PREM scheduling, and thus depends on what form of scheduling is used.

Fundamentally, this function is used to call the GPUguard synchronizations when executing on the GPU, triggering the synchronization process outlined in Section 2.1.3. As we will show in Section 4.2, this function can also be used to enforce a predefined static schedule in CPU-only execution. In this case, the scheduler uses the unique identifiers for each interval generated in the DAG for the offline scheduling, and uses the same identifier `id` passed through the library function call to enforce the order at runtime. The `type` argument is used to separate the phases from within the labeled intervals, and thus corresponds to either *prefetch*, *compute*, *writeback*, or *compatible*.

The semantics of the function is that once it returns control to the PREM-transformed program, it is safe according to the PREM schedule to continue the execution of the following phase (identified by the `id` and `type` arguments), in accordance with the discussion in Section 3.1.3.

In addition to the `__prem_notify()` function, the library implements `__prem_init()` and `__prem_fini()`, calls to which are inserted by the compiler at the very beginning of the control flow (the entry point as described in Section 3.2.2) and at the end points of that function. This function can be used by *libprem* to initialize internal structures needed for the scheduling, if needed.

This separation of concerns into an easily customizable library makes it possible to schedule PREM tasks generated with the compiler on diverse platforms with diverse types of scheduling, as we will show in the remaining chapters of this thesis.

3.3 Discussion and Concluding Remarks

The compiler presented is able to transform programs written in accodance with real-time coding guidlines into PREM applications. The compiler produces the necessary runtime hooks and scheduling information required to construct a full working PREM system. We will evaluate this in Chapter 4.

Before the evaluation, we look into the limitations that have been identified throughout this chapter, and discuss the impacts of these limitations, as well as promising ways to lift them.

Translation Units and PREM transformations One of the most prominent limitations, identified in Section 3.2.2, is that the PREM compiler is not able to analyze functions that lie outside the current translation unit. This corresponds to external functions from libraries or syscalls, but also to functions and symbols implemented in another source file than the one currently being processed. The former is a predicted limitation of PREM, as outlined by Pellizzoni et al [24] and originally motivating compatible intervals. The latter is a limitation that follows from the way compilers operate, but that was not foreseen in previous PREM publications.

This limitation can easily be worked around by ensuring that the entire program – or if a sub-set, the parts that are to be PREMized – are part of the same compilation unit. For example, if a program consists of files `A.c` and `B.c`, they can be included in the same compilation unit by creating file `AB.c` that contains the lines `#include "A.c"` and `#include "B.c"`. In doing so and compiling `AB.c` the compiler will have full visibility of the symbols and functions in both files simultaneously, and the problem is averted.

The issue could potentially be worked around by keeping a persistent cache of the analysis results of functions in previous compilation units, and ordering the compilation of files such that if a file A uses a symbol defined in file B, B is compiled before A, and the analysis results stored to disk. At the compilation of A, instead of marking the symbol from B as *unpredictable* the relevant information from the cache could be loaded from disk. This would impose further requirements on the compilation process: Only after all files have been analyzed the interval selection can be performed, only after which the transformations can take place. Thus, the transformation instructions would then have to be written to disk, and the compiler process restarted anew to apply the changes as they correspond to each translation unit individually. An investigation on what limitation this multi-translation unit transformation would impose on how intervals are slected (and can be transformed) would be a necessary first step to determine if this is suitable, or if one would rather enforce that all aspects of the task to be PREMized is part of a single translation unit.

A Broader Sense of Compatible Intervals In Chapter 1 we outlined that compatible intervals were proposed in the original proposal of PREM [24] to address non-analyzable syscalls in legacy code. This served as a way to not impose unrealistic limitations on programs that execute in a PREM system. In our work we have extended the understanding of compatible intervals to include *any* piece of code that the compiler could not sufficiently analyze to generate three-phase predictable intervals.

This extension is advantageous, because it allows any program to be compiled with the PREM compiler, even though parts of it does not adhere to the requirements for PREM. This enables programmer-in-the-loop decision making on the correct course of action. If the cost of rewriting the code is too high (in time, performance, or for other reasons) in relation to its impact on the PREM schedulability, then a system designer may be better served to allow this code region to remain a compatible interval. If this region is critical for PREM schedulability or performance, then the designer can selectively invest their effort in these cases.

As outlined, there are several requirements imposed to code that is to be PREMized, and producing a binary only if every requirement is fulfilled in every line of code may be overly limiting. We therefore argue that this extended understanding of what code that is best served as compatible intervals allows for more flexibility and applicability of PREM as a whole. The proposed understanding of compatible intervals outlined in this Chapter are a strict superset of what was proposed in [24].

Larger is Not Always Better? Based on the findings of Chapter 2, the current compiler optimizes for PREM intervals as large as possible, to ensure that the overhead of frequent scheduling points can be amortized over longer phases of useful work. However, this optimization is concerned only with the performance of a single task. As PREM fundamentally comes down to scheduling the memory phases such that they are non-overlapping, longer phases provides the scheduler with coarser granularities, which may lead to less optimal final schedules.

However, within the compiler such optimizations are not possible, as the compiler is once again limited to the visibility provided by the translation unit. Thus it cannot identify opportunities to perform better interval selections that can lead to better schedules. However, due to the importance of this aspect it will be further discussed in Chapter 6 where we present an extended PREM toolchain capable of triggering interval selections based on feedback from the scheduler. This enables interval selection to optimize beyond the scope of the compilation unit, and across the entire system.

Symbolic Scalar Evolution Analysis As outlined in Section 3.1.3, Scalar Evolution analysis may return symbolic expressions for the evolution of scalar variables. If such variables are used to index into arrays, the implemented version of the compiler is unable to determine the footprint of the analyzed loop. The simple solution, and indeed one often used for static analysis in the real-time domain, is to enable these symbolic expressions to be estimated by providing programmer-annotated bounds, i.e., using `#pragmas`.

As such, the static footprint analysis implemented in the compiler is subject to the same limitations for loop analysis that any other static analysis tool is, and the same solutions can be used to work around them. For affine loops it is likely possible to work around this issue by assuming that each symbol in a symbolic loop expression is large enough to make the overall loop is larger than the local memory used for PREM, and enforce tiling based on this assumption. At that point, no matter what the actual runtime values of the symbols are, the loop is pre-tiled to ensure safe execution of the corresponding intervals. This provides an interesting step for future development of the compiler.

Optimal Tiling and SPM Buffer Management Tiling and SPM Buffer Management individually comprise separate fields of research. In this work we have not aimed at extending the state of the art in either of these aspects, but used previous approaches presented in the literature, e.g., tiling [68, 75] or implemented representative proof-of-concepts, e.g., buffer allocation. This was motivated by the focus on exploring the possibility of constructing a PREM compiler, rather than a specific interest in either of the already understood components.

Due to the fundamental requirement on static analysis within a PREM compiler, it provides an interesting use-case for novel techniques in these domains, as they can be constructed to rely on more information than what can typically be expected from a general purpose workload. In addition to this, the PREM compiler in itself would benefit from more advanced techniques to perform better optimized tiling [66, 67] and SPM allocation [32].

On Pointer Analysis Pointer analysis remains an important problem for any compiler analysis and transformation, so also for PREM. Even though real-time coding guidelines discourage or even forbid code constructs where such analysis is made more difficult – MISRA C [62] for example forbids nesting of pointer arithmetics except for array indexing – there is still software out there that relies on such constructs. A typical example would be graph traversal algorithms, stored in linked lists or similar structures. Such structures are not supported by the presented PREM compiler, and due to the difficulty in analysing such structures without traversing them at runtime, it is unlikely that they ever will. Some approaches, such as runtime prefetcher threads [76] could potentially lift this issue, but would require extended runtime techniques. As is now, all such structures are transformed to compatible intervals, with the corresponding compiler warning to the user.

Code Prefetching The presented techniques in this chapter relate solely to the prefetching of data for each PREM interval. The compiler is trivially extendable to prefetch also code: As each PREM interval consists of outlined functions, the prefetch phase of the interval could be extended with a prefetch call on the address of the compute phase function. The only additional support required would be to, at the end of the code generation step, fill in the length of the generated instruction stream for the function.

3.3.1 Conclusion

This section has presented the first large-scale attempt to develop a compiler capable of transforming legacy code to be compatible with the three-phase requirement of PREM. This provides an important step to understand how feasible the PREM approach is in practice, as manual refactoring is an error prone process, while compiler automation imposes certain limitations on the code.

Our conclusion is that PREM will imply certain requirements on programmers of embedded systems, but that with these taken into account, the most significant limitation of PREM itself, the code refactoring requirement, can be automated. This removes one of the large hurdles to making PREM a feasible approach for future embedded real-time systems.

In the next section we will evaluate the compiler with real benchmarks running on real systems, which is the first such experiments with PREM.

Chapter 4

Co-scheduling Heterogeneous Systems with Freedom from Interference

In Chapter 2 we showed that the freedom from interference guarantees that PREM promises can be realized in practice on heterogeneous platforms. At that point the evaluation was limited to a small set of manually PREMized programs, as refactoring code to align with the three-phase PREM structure is a tedious task. In Chapter 3 we showed how a compiler can be implemented to automate this task, for platforms with different characteristics.

In this Chapter, we will use the presented compilers to test PREM on a broader range of platforms, and with a broader set of workloads. It explores the performance impact from the compiler transformation and runtimes, as well as the ability of the transformed programs to achieve the freedom from interference guarantees that motivate PREM. In doing so we will identify challenges that will be addressed in the following chapters.

We begin in Section 4.1 with a deep evaluation of a large set of benchmarks on the same platform used in Chapter 2, the NVIDIA Jetson embedded GPU systems. Following this Section 4.2 presents the extended compiler presented in Section 3.2 coupled with optimal PREM scheduling on ARM CPUs. Lastly, Section 4.3 presents our findings when deploying PREM on PULP, a programmable many-core accelerator with access to large SPM-based tightly coupled data memories.

4.1 Compiler-generated GPU programs

In this section we begin by exploring the impact of PREM on the NVIDIA TX1 GPU, for benchmarks compiled with the compiler presented in Section 3.1. This section presents results published at *DATE 2018* [57] and in the *IEEE Journal Transactions on Computers* [40].

The experimental platform and the PREM scheduling parameters follow in Section 4.1.1, which provides the necessary background information to the experimental results. This section has three goals, which will be addressed as follows: 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 Section 4.1.2, explores the performance impact of code transformation, synchronization, and changes in memory access patterns. Here we compare the novel SoftDMA (SDMA) both to compatible intervals (CMPT) and predictable intervals achieved with the previously presented stripping techniques, coinciding with the DAE technique [68] (DAE). The second block, in Section 4.1.4, discusses predictability results for *SoftDMA* compared to the baseline OpenMP implementation without PREM.

4.1.1 Setup

We begin by presenting the NVIDIA TX1 on which the evaluation is performed. Following this, this section provides the PREM scheduling paradigms that will be explored. This provides the necessary background information for the experimental results that follow in the coming sections. In this section we are primarily interested in the

GPU.

Platform We utilize the compiler presented in Section 3.1 to generate code for our experimental platform, the NVIDIA TX1. For this purpose, we use LLVM's NVPTX (NVIDIA Parallel Threads eXecution) backend [64, 65, 77] to generate code for the platform. 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. This system is well-aligned with the platform model presented in Section 1.1.1, and due to the shared DRAM between the CPU and the GPU the platform is susceptible to memory interference.

As a representative GPU workload we consider kernels from the PolyBench-ACC suite [78], compiled with the *standard* data set for the best performing block/grid dimensions.

The *GPUguard* 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.

PREM co-scheduling configuration Following from the discussion in Section 2.1.2, to produce the results in this section the system is scheduled using fixed-size *quanta* (Section 2.1.2) for the CPU and the GPU, 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 $p_{memory} + p_{compute} = 100\%$. In this section we refer to p_{memory} and $p_{compute}$ as the *scheduling parameters p*. These paramters can be freely configured and always produce a valid PREM schedule, as will be shown in Equation 4.1. We have already ensured that the fundamental PREM property of Equation 1.1 are upheld from the compiler. The remaining thing to ensure is that the scheduling parameters p create time quanta E such that they are always long enough to contain the full worst-case execution time

T of the phases. This is achieved through the relationships expressed in Equation 4.1.

$$\begin{aligned} E_{memory} &= \begin{cases} T_{memory}, & \text{if } \frac{T_{compute}}{T_{memory}} \leq \frac{p_{compute}}{p_{memory}} \\ \frac{p_{memory}}{p_{compute}} \times T_{compute} & \text{otherwise} \end{cases} \\ E_{compute} &= \begin{cases} \frac{p_{compute}}{p_{memory}} \times T_{memory} & \text{if } \frac{T_{compute}}{T_{memory}} \leq \frac{p_{compute}}{p_{memory}} \\ T_{compute} & \text{otherwise} \end{cases} \end{aligned} \tag{4.1}$$

Thus, the execution time of an interval, with the system schedule taken into account is $E = E_{memory} + E_{compute}$, which thanks to Equation 4.1 is always large enough to allow the phase to finish. Any slack time that remains at the end of the quanta, i.e., due to execution finishing earlier than its WCET or overbudgeting due to p, will introduce idle time I into the system, which we will quantify empirically as part of our evaluation. Theoretically, the idle time I can be determined as shown in Equation 4.2.

$$I = (E_{memory} + E_{compute}) - (T_{memory} + T_{compute}) \tag{4.2}$$

When the scheduling parameters p are aligned with the phase execution times T such that Equation 4.1 gives an E such that $I = 0$, we say that we have a *best-case* schedule. Note however, that as T is the worst case execution time, the system may still idle if the actual execution time is lower. We refer to this later type of idling as budget idling I_{budget}, which is given for each execution by the difference between the WCET T of the task and the actual execution time t_i of the ith executed instance of the phase.

4.1.2 Performance Evaluation

We begin by experimentally testing the performance of the PREM-compatible GPU kernels created by the compiler. Due to the code transformations and runtime scheduling decisions we expect this to have an impact on the performance. In particular, we expect the performance to be negatively affected by the scheduling through GPU-guard synchronziations, while the tiling and use of the SPM can have a positive effect on performance due to increased data locality. In

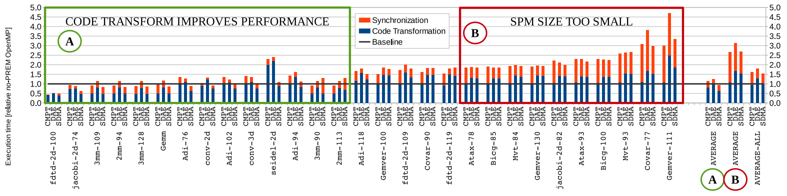


Figure 4.1: The performance of code transformation and synchronization on the kernels, relative to the unmodified kernels.

this section we therefore compare the performance of the PREM-transformed kernels to that of the standard OpenMP kernels without PREM transformations. In the following Section 4.1.4 we will then compare if PREM is able to maintain this level of performance when exposed to memory interference, and if overheads outlined here can be amortized by improved WCET.

We begin by exploring the impact on the three main transformation styles presented in Section 3.1.4, *compatible*, *stripped*, and *SoftDMA*. In the first instance, we will discuss only the fundamental stripping technique, here illustrated by the technique equivalent to Decoupled Access Execute (DAE) [68].

We configure the system and compiler as outlined in Section 4.1.1, execute the kernels on the GPU, and present the results in Figure 4.1, highlighting the effect of PREM code transformations and synchronization on the execution times of the kernels. All execution times have been normalized to that of the unmodified OpenMP baseline, and all reported timing results are the *measured* worst case execution times (WCET). For all performance-related experiments we measure execution time in isolation (i.e., without memory interference).

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. We will discuss the impact of these generally over the set of benchmarks now, and further extend our understanding of the performance impact on individual benchmarks in Section 4.1.5.

Instruction count and access patterns

Both DAE and SoftDMA 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, as can clearly be seen in the benchmarks in the left side of Figure 4.1 (labeled A). 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 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.

Synchronization

Together with the effects of code transformation, the *synchronization* overhead is also presented in Figure 4.1, illustrated by *the red part of the bars*. This synchronization cost is due to the GPUguard token passing to ensure that the PREM memory isolation property is upheld for the compiled programs at runtime. In contrast to the GPUguard prototype presented in Chapter 2, in these experiments we deferred the throttle thread wakeup to the lower half of the interrupt handler, shortening the synchronization latency compared to Figure 2.4 to $S_{measured} = 5.8\mu s$, without loss in predictability as will be discussed in Section 4.1.4.

As outlined in Section 2.1.2 the absolute synchronization cost S is equal for all kernels, and its impact on each kernel is determined by how well S is amortized with useful work T. 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 Figure 4.1 (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.

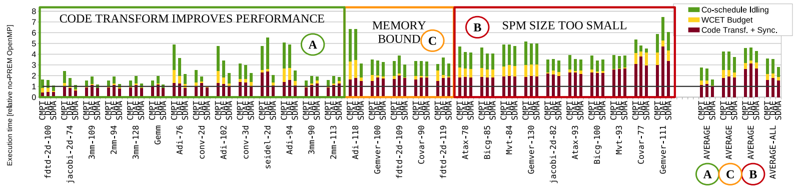


Figure 4.2: The idling introduced into the program due to enforcing the *Fair* system memory schedule.

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[1] (including synchronization cost $S_{measured}$). 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).

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, as we will discuss further in Chapter 5.

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, i.e., compatible, stripping and SoftDMA.

4.1.3 PREM Scheduling Effects Evaluation

PREM requires enforcing the WCET for each phase before triggering the synchronization that precedes the beginning of a new phase. In

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

our evaluation 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.

Under these conditions, Figure 4.2 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 Figure 4.1. 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. This corresponds to I_{budget} as outlined in Section 4.1.1. 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, introducing non-optimal quantas $E > T$ and the corresponding idling I as defined in Equation 4.2. Lastly, in addition to the A and B categorizations introduced in Figure 4.1, Figure 4.2 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

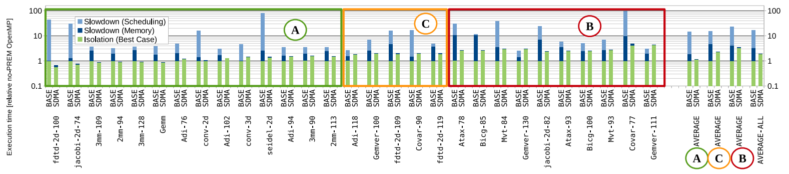


Figure 4.3: The performance degradation due to memory interference from the CPU for the *best case* system schedule.

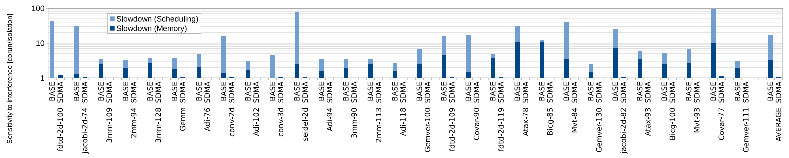


Figure 4.4: The sensitivity to interference for the BASE and SDMA versions of the kernels.

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 Figure 4.2.

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 Section 3.1.4. 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. Figure 4.2 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 Figure 4.2 (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-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. 4.1 gives $E \approx T$, which introduces a low amount of idling I as given by Eq. 4.2. 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 Figure 4.2, 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.

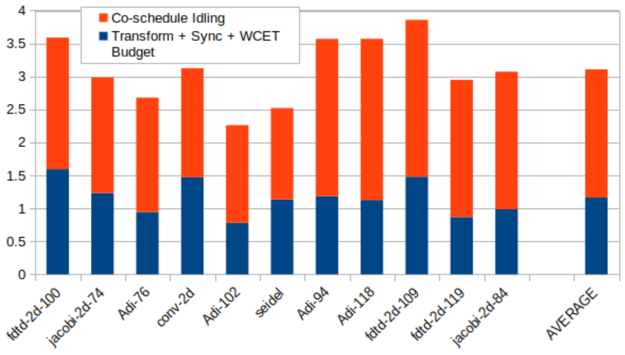


Figure 4.5: The execution times relative to the non-PREM OpenMP baseline for *Combined* kernels under *Best fit* and *Fair* scheduling.

SoftDMA and Stripping Techniques in Comparison

There is one additional Stripping technique that we have not yet evaluated. This is the *Combined* scheme described in Section 3.1.4.

The motivation for the Combined scheme is the same as for SoftDMA; the standard DAE transform makes insufficient well use of the memory bandwidth, and as such introduces significant overhead in the transformed PREM code. While SoftDMA addresses this problem by generating memory phases with the minimum amount of memory accesses, as shown in Figure 3.6, the Combined scheme instead constructs memory phases to perform in-memory computation of tile $n+1$ while data is loaded into the local memory for tile n. This means that two iterations are handled per PREM interval. This was our first attempt at solving the underlying bandwidth utilization issue, and was presented in the *DATE'18* publication [57], and later replaced by SoftDMA in the *Transactions on Computers* [40] publication.

We compare the performance of the transformed code[2] under the *Best fit* schedule described previously in the blue parts of the bars for Combined in Figure 4.5 and Figure 4.6 for SoftDMA. Both of these approaches solve the fundamental problem of memory bandwidth uti-

[2]The comparison is limited to the kernels that were used in the evaluation in the *DATE'18* publication [57].

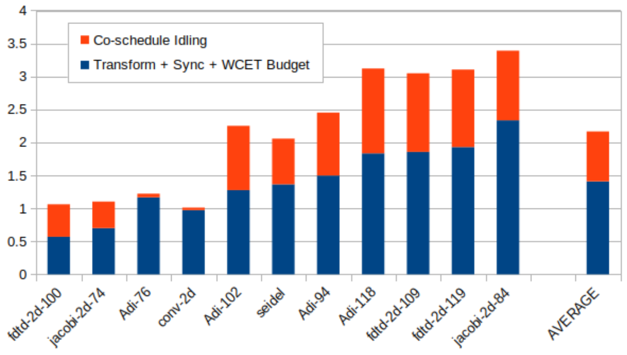


Figure 4.6: The execution times relative to the non-PREM OpenMP baseline for *SoftDMA* kernels under *Best fit* and *Fair* scheduling.

lization, and achieve similar performance when measuring transformation and synchronization costs – the Combined scheme is even performing a bit better on average, although for individual benchmarks that benefitted from the data-reuse aware transformation of SoftDMA the performance can be significantly worse. The lower average comes from not being as sensitive to the synchronization wall effects, due to longer phase execution times for the two-tile execution which amortizes the synchronization costs, and switching phases less often than limited by the maximum synchronization frequency.

The big difference comes when comparing how they perform under co-scheduling with the CPU, using the *Fair* scheme described previously. As already outlined, the DAE/Stripping approach, upon which Combined is built, is doing significantly worse in moving execution time from the memory phase T_{memory} to the compute phase $T_{compute}$, leading to significant idling in the compute phase. This effect is increased when the memory phase is further overloaded to perform an in-memory computation at the same time, leading to even more idling as T_{memory} increases. As discussed in the previous section, SoftDMA on average reduces the idling compared to DAE by 45%, however, as can be seen in Figures 4.5 and 4.6, when comparing to the Combined scheme, SoftDMA is able to reduce the idling by over 60%.

Thus, even though the combined scheme does address the memory bandwidth utilization issue, and may perform slightly better than SoftDMA on average (in the *Best fit* schedule), it can perform significantly worse for kernels that do not hit the synchronization wall, and introduces significant amounts of idling in *Fair* scheduling. For this reason, following the development of SoftDMA, we dropped all further investigations into the Combined scheme, as will also be the case in this thesis.

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× slowdown in CMPT, and 2.67× in DAE. iii) In heavily memory bound kernels, the inability to perform work in the compute phase increases the average slowdown to 3.8× 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×. This happens when the SPM is too small, and is an effect of the hardware used and not the technique itself.

4.1.4 PREM effects on Predictability

For predictability results we measure execution time in presence of high memory interference. The memory interference from the CPU is generated using the *stress* [55] tool, which is able to produce large amounts of memory interference on a system.

To validate the effectiveness of the proposed SDMA toward guaranteeing *robustness to interference*, we execute the the execution of all GPU kernels under heavy interference from the CPU, as outlined in Section 4.1.1. The results for the *best case* scenario are shown in Figure 4.3. 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 Section 3.1.1 that the interference consists of two parts, *memory interference* and *scheduling jitter.* While SDMA is not affected by the latter (due to GPUguard protection), 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 Figure 4.4, 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, to this, the Predictable Execution Model [28] provides *robustness to interference* by design, which means that near-zero interference is indeed the expected value.

4.1.5 Performance Estimation at Compile-Time

Having concluded that the compiler-transformed GPU kernels coupled with GPUguard indeed fulfil the PREM freedom from interference guarantee, we return to the question of PREM performance. As shown in Section 4.1.2, the performance differs significantly between the presented benchmarks, which we now explore in detail. Our overarching goal is to make predictions on the performance from information that is or could be made available to the compiler, in order to predict performance and steer compiler choices. This work is focused on the SoftDMA transformation for GPU kernels, and was published at *SCOPES'18* [79]. The performance impact is determined by both hardware characteristics as well as characteristics of the benchmarks themselves.

The hardware characteristics we found useful to predict performance are the latencies of (a) arithmetic operations $l_{arithmetic}$, (b) scratchpad accesses l_{SPM}, and (c) DRAM accesses l_{DRAM}. The average latency of arithmetic operations $l_{arithmetic}$ is reported by NVIDIA as 11 cycles [80], and the latency for accessing the scratchpad is reported by Mei et al. [81] as 28 cycles on the TX1 Maxwell architecture. The DRAM latency was determined using the synthetic benchmark presented in Section 2.1.4 where the GPUguard prototype was evaluated. We extend the synthetic benchmark to not only consider the number of threads used for accesses, but also the memory access pattern – in Chapter 2 we were only concerned with the smallest latencies, to determine the smallest period at which synchronization would have to occur. To predict the performance of any benchmark the scope has to be extended to comprise two cases. First, when memory accesses are *coalesced*, i.e., threads executing concurrently will touch data on consecutive addresses, and second when the accesses are sparse, i.e., every thread will need to fetch a unique cache line to satisfy its memory request. The results are shown in Figure 4.7, where the coalesced curve corresponds to Figure 2.4 in the GPUguard prototype evaluation. As can be seen, the latency for memory accesses that are coalesced and sparse differ by an order of magnitude, and thus need to be handled separately. We therefore further divide l_{DRAM} into the latency for coalesced memory accesses $l_{DRAM}^{coal}$and the latency of sparse memory accesses l_{DRAM}^{sparse}.

The latency for the operations would have to be included in the compiler for each supported platform, some support for which is already available in the instruction scheduler in the compiler backend. The benchmark-specific characteristics can be extracted from the code under compilation by mapping each instruction to one of the identified hardware latency classes.

Due to the loop-based offloading in high-level languages, the compiler is able to distinguish coalesced from sparse accesses to arrays through scalar evolution analysis, as each value of the induction variable (IV) of the offloaded loop maps to a specific thread[3]. By ana-

[3]Assuming static scheduling, which we committed to in Section 3.1.1.

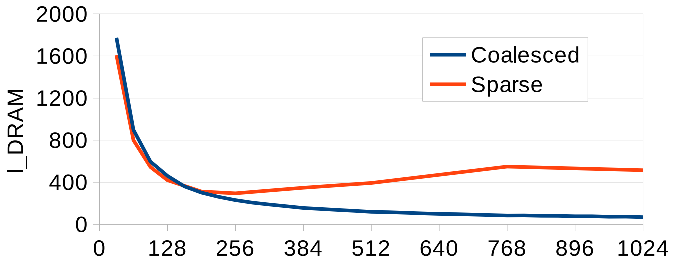


Figure 4.7: The difference in time it takes to load fully sequential and non-sequential data into the scratchpad (49152 bytes).

lyzing the evolution of the IV it is thus possible to determine which thread will perform each access, and by extension, if the access is part of a coalesced pattern. The compiler can therefore count the number of arithmetic operations C, and the number of coalesced M_{coal} and sparse M_{sparse} memory accesses. These operations map to the identified latencies of $l_{arithmetic}$, $l^{coal}_{DRAM}$and l^{sparse}_{DRAM}.

For the SoftDMA transformed code, these accesses will also result in accesses to the SPM during the memory phases, at the corresponding l_{SPM} latency. The benefit is that DRAM will only be accessed once per datum even if it is used multiple times, as the SPM provides local low-latency accesses. Thus, the compiler also keeps track of the unique memory accesses U_{coal} and U_{sparse} – this information is implicitly provided by the access map A presented in Section 3.2.2.

4.1.6 Modeling the Execution Time

When estimating the performance of the SoftDMA transformations, both the cost of executing the PREM phases T_{Phases}, and the time required for the synchronization T_{Sync} must be taken into account. As the SoftDMA code consists of two parts, T_{Phases} is further split up into the individual execution times for the Compute and Memory phases such that $T_{Phases} = T_{Memory} + T_{Compute}$.

The memory phase execution time is dependent on both the data movements from DRAM, as well as the cost for accessing the scratchpad memory through which the data is staged. Because of this, the latency of both memories is taken into account when modeling the memory phase. As already outlined, the SoftDMA memory phases imply that data is loaded only once from DRAM, even if reused, and as such only the unique accesses U are considered. The latency of arithmetic instructions is only considered in the Compute phase, and as its accesses are performed on the scratchpad memory, only these latencies need to be considered. From this, we model the execution time of each phase as shown in Equation 4.3.

$$\begin{aligned} T_{Memory} &= U_{coal} \times l_{DRAM}^{coal} + U_{coal} \times l_{SPM}^{coal} + \\ &\quad U_{sparse} \times l_{DRAM}^{sparse} + U_{sparse} \times l_{SPM}^{sparse} \\ T_{Compute} &= C \times l_{arithmetic} + M_{coal} \times l_{SPM}^{coal} + \\ &\quad M_{sparse} \times l_{SPM}^{sparse} \end{aligned} \tag{4.3}$$

The cost of performing the synchronization S with the host at the end of each PREM interval is a system-dependent parameter, found in Section 4.1.2 to be $5.8\mu s$ for the NVIDIA TX1. Importantly however, we saw in Section 4.1.2 that some kernels suffer additional overheads because of the minimum synchronization granularity, i.e., the *synchronization wall*. We therefore include the minimum synchronization granularity R_{timer} ($10.7\mu s$ on the TX1) at which the timers can be triggered, and thus each phase is forced to execute for at least this time (idling if the phase is shorter). The full SoftDMA interval execution time is as shown in Equation 4.4.

$$\begin{aligned} T_{SoftDMA} = max(T_{Memory}, R_{timer}) + \\ max(T_{Compute}, R_{timer}) + \\ 2 \times S \end{aligned} \tag{4.4}$$

The modeled execution time is validated against measured values from Section 4.1.2, and the results are presented in Figure 4.8. It can be seen that the predicted values follow quite accurately the measured ones (the error is on average below 10%). From this we conclude that the hardware and benchmark characteristics that we have identified in Section 4.1.2 and reidentified here are indeed able to explain the performance of the SoftDMA transformed kernels. We also conclude

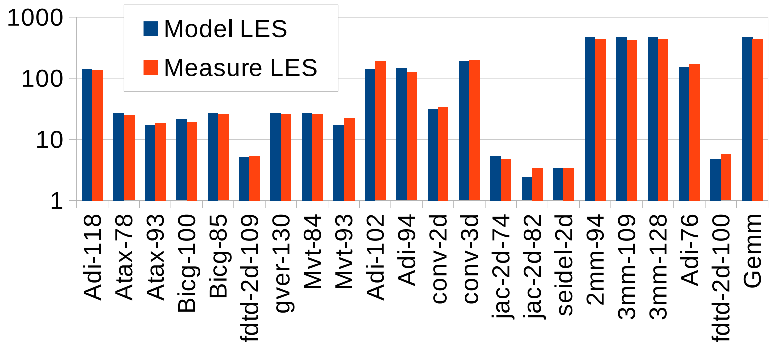


Figure 4.8: Validation of the modeled LES execution times against measurements on real hardware.

that it would be feasible to implement such predictions within the compiler to steer compiler choices, by encoding the arithmetic and memory latencies, as well as the platform-specific synchronziation cost and the minimum granularity at which it can be performed.

However, the main motivation for implementing such predictions in the compiler is to support platforms where the local memory, e.g., the SPM, is small, to catch the cases where the overheads for PREM transformations are significant. In the following sections we will present the corresponding transformations made on platforms with larger memories, and show that the overheads in those cases are not large enough to warrant such concern. Following this, in Chapter 5 we will show how the larger hardware-managed caches can be used to achieve similar low-overhead execution on the GPU.

4.2 Extended CPU Compiler with PREM Scheduling on ARM CPUs

Following the GPU experiments in the previous section, we now concentrate on the evaluation of the extended version of the compiler, presented in Section 3.2. The extended compiler uses region analysis to drive the PREM transformations for workloads much more general than the GPU-only compiler presented in the previous section. For that reason, this evaluation is focused on the ARM CPU.

The work presented here was done in collaboration with Joel Matějka at CTU Prague, and resulted in a conference publication at the 9th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM) [82] with a journal extension published in Elsevier's Journal on Parallel Computing (PARCO) [36]. The contributions of those publication was twofold. First, to demonstrate PREM compilation for CPU, and second, to present a new PREM scheduling methodology, based on an ILP solver and a Heuristic version of the same. Here we are primarily interested in the former part, i.e., the compiler, but we will present results that also use the mentioned scheduler, based on work by Hanzálek et al [83], to show the freedom-from-interference results achieved on general purpose ARM cores. Furthermore, GPUguard used for PREM scheduling in the previous two sections is able to manage the PREM scheduling across the CPU-GPU boundary, but not within the CPU – making further case for the use of custom PREM scheduling techniques.

4.2.1 Cache Considerations

For this evaluation we use the CPU complex of the NVIDIA TX1 platform. It consists of four ARM A57 cores, each core with a 32KB L1 cache, and a shared L2 last level cache at 512KB. Our experiments in this section does not consider any GPU kernels, but only parallelism within the CPU complex.

The ARM CPUs of the NVIDIA TX1, in contrast to the GPU, do not have access to a scratchpad memory, but data locality is instead provided by hardware managed caches. These caches employ a random replacement policy which makes it difficult to ensure that data prefetched in the PREM prefetch phase remains in the cache until the compute phase. Caches are subject to three types of misses; capacity misses, mandatory misses, and conflict misses. Capacity misses are avoided in PREM as the memory footprint of each interval is smaller than the cache (see Section 1.4). Mandatory (cold-start) misses are not an issue either, as data is explicitly prefetched during the memory phase. Thus, only conflict misses need to be addressed.

Cache lines are stored into *sets*, based on the *index bits* of the address. As these benchmarks predominantly store data in arrays, the associativity gives the number of times the *index bits* of a single array can wrap before the sets are exhausted. With multiple arrays they compete with each other for these sets. The problem is that all cache conflicts on the A57 are resolved by the Random Replacement Policy (RRP) [36, 37]. To illustrate the problem, consider any current interval i_n which fetches data d_0^{new} and d_1^{new}, and a previous interval i_{n-1} that has fetched d^{stale}. Here, d_0^{new}, d_1^{new}, and d^{stale} index into the same cache set, and we assume the associativity is two, meaning two data can be stored in that set. At the beginning of i_n, the datum d^{stale} of the previous interval i_{n-1} is still in the cache, and once d_0^{new} is fetched, the cache contains d_0^{new} and d^{stale}. The problem occurs when d_1^{new} is fetched, as this causes the RRP to randomly evict either d^{stale} or d_0^{new}. The former is OK, but the latter would lead to a cache miss for d_0^{new} during the compute phase of i_n, violating the PREM isolation guarantees.

Luckily, the cache prefers to place data in *invalid* cache ways before evicting data [37]. This can be leveraged under PREM, as the *writeback phase* can invalidate the data of i_{n-1} before i_n begins, thus removing d^{stale} from the cache before i_n starts, and side-stepping the RRP. We call this mechanism *preventive invalidation*, showing that the PREM *writeback* phase, envisioned for SPMs, is also necessary for RRP caches.

For this reason, we configure the compiler to create writeback phases that contain the `dc civac` instruction, which evicts the specific cache line from the cache, freeing it up for reuse, and importantly, avoiding that other cache lines are evicted. However, due to the low size and associativity of the L1 cache the shared L2 was used for these experiments, which could lead to evictions from code running on other cores. In this evaluation we work around this limitation by utilizing only half the size of the L2 PREM intervals, and as we will show in Section 4.2.5 this allows PREM intervals to execute without significant cache misses. A further discussion on this will follow in Section 5.2, where we utilize cache coloring to ensure isolation also between different cores.

4.2.2 Kernels

In this evaluation a reduced set of benchmarks were used in comparison to the previous evaluation – which already gave us a good understanding of their individual characterisitics – but instead we combine these kernels into difference *scenarios*, in which different benchmarks are co-scheduled with each other. Following from this no external source of interference is required, as the different co-scheduled benchmarks will interfere with each other. However, for consistency we will do such experiments as well.

We consider five kernels, matrix multiplication *gemm mul* and transposition *gemm tran*, 2D convolution *2Dconv*, 2D Jacobi stencil computation *2Djacobi*, fast Fourier transform *fft*, and a binary tree search *bts*. These kernels have different compute to communication (CCR) ratios and different access patterns, and thus represent different types of memory behavior, an important aspect to consider as outlined for the GPU in Section 4.1.2 and 4.1.5. The *2Djacobi* program consists of two kernels, where the second kernel *2Djacobi-2* is a data copy kernel – i.e., it has no computation. The *bts* benchmark, due to its graph traversal computation cannot be efficiently turned into PREM intervals, as discussed in Section 3.3, and as such is represented by a compatible interval.

We begin by PREMizing and measuring the performance of the individual kernels comparing it to the original non-PREMized version. This corresponds to the evaluation done for the GPU in Section 4.1.2. The results are presented in Figure 4.9. Following this, we rerun the same experiment under memory interference to characterize the sensitivity to intererence for all benchmarks, showing the results in Figure 4.10. In these experiments, we do not perform a specific experiment for the synchronization overhead, as it is negligible due to two reasons. First, due to the larger caches the synchronization occur much less frequently, and second, the passing of the schedule through a predefined sequence within the CPU removes the need for costly interrupt handling to coordinate memory transactions with the GPU.

Beginning with the performance characterization, we see that when only considering the prefetch and compute phases, PREM is able to achieve on-par or even improve performance for all kernels. This is

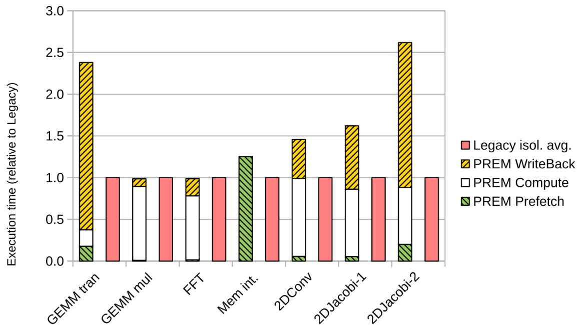


Figure 4.9: Normalized execution times of kernels.

in line with our observations on the GPU, and are similarly due to the improved data locality, and the improved memory bandwidth of the prefetch phases – prefetches are generated at the maximum speed possible, as no computation takes place in between. However, we see that the writeback phases, i.e., preventive invalidation, add significant overhead on top of this. The impact is proportional to the CCR of the benchmarks, as benchmarks with higher CCR are dominated by the length of the compute phase, amortizing the cost of preventive invalidation. The legacy programs, which do no cache house keeping do not suffer from this overhead, meaning that PREM execution suffers up to 2.6× overhead, as shown for the copy kernel *2Djacobi-2* (which has a CCR of zero, due to lack of computation) for this requirement. However, as we will see in Section 4.2.5, even in light of such overheads[4] the PREM system overall can still outperform a legacy system, even if such tasks are part of the taskset.

Considering the kernels under interference, as shown in Figure 4.10, we can see two important things. First, the PREM execution times remain unchanged and achieve the freedom from interference promised. Second, the legacy code execution times increase drastically, in some cases up to over 2.6× their execution times in isolation. The 2.6× overhead previously reported for the PREMized version of *2DJacobi-2* shrinks to only 50% when compared to the execution

[4]As we will explain in Section 5.2.1, these overheads can be further reduced.

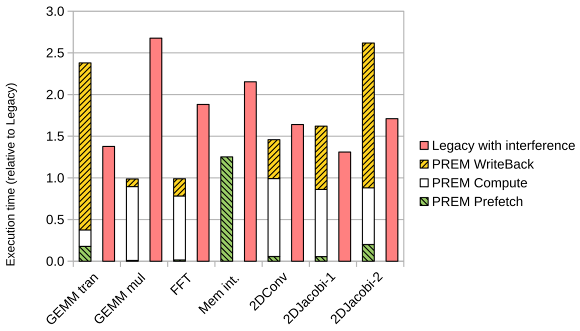


Figure 4.10: Normalized execution times of kernels under external memory interference.

time of the legacy version under interference. In many other cases the legacy execution time increases so much that the PREM version outperforms it, even in light of the overeheads introduced due to cache bookkeeping.

4.2.3 Use-case scenarios

Having understood the effect of memory interference and PREM transformations on the individual kernels, we now proceed to create *scenarios* of task sets, that we use the techniques presented in the papers by Matějka, Forsberg, et al [82, 36] to schedule. These scenarios allow us to test how PREM behaves over a full system of tasks that are co-scheduled on multiple cores, as opposed to only its effects on individual kernels. To test scheduling with both small and large scenarios, two classes were created. The first class is that of a small amount of PREM intervals, that can be scheduled optimally using the ILP solver-based solution from CTU Prague [36]. The second class has a large amount of tasks, the optimal scheduling solution to which is intractable with ILP solvers. This class whs instead scheduled with a heuristic approach, also developed at CTU Prague [36].

To understand how tasks/kernels contribute to the scenarios, an example is provided in Figure 4.11. The scenario contains three DAGs, each representing one application to be co-scheduled, and vertically the DAGs are divided to represent the different benchmarks that it consists of. The edges of the DAG represent dependencies, i.e., the interval with the outgoing edge needs to finish before the one where the edge is incoming can begin. Following from this there are two types of parallelism expressed in the DAG. First, each individual subgraph represents parallelism, as their execution does not depend on the completion of each other. Second, within each task fork-join style parallelism is employed (e.g., OpenMP) and is exposed to the scheduler as parallel executions that do not depend on each other, e.g., in *gemm mul*. Each individual benchmark consists of intervals I, which are enumerated across the entire application (e.g., it does not restart for each individual DAG or benchmark), making it possible to individually identify each interval within the system. The figure shows both predictable and compatible intervals. Predictable intervals consist of a prefetch (red), compute (white), and a writeback (also red) phase, while compatible intervals consist only of a single phase (green). The scheduler will schedule the prefetch and compute intervals together, while the writeback phase can be deferred, as illustrated by the arrow between the compute phase and writeback phase.

This particular scenario consists of three parallel tasks, starting with intervals I_1, I_{10}, and I_{12} respectively. The first task consists of the *gemm tran* and *gemm mul* kernels, repeated twice in different configurations, and ending with a compatible interval to rejoin the threads. Within the *gemm mul* task further parallelism is exposed. The second tasks consists of two chained *fft*s, and the final task of five chained *bst*. The scenarios considered for the evaluation are as follows:

Four small scenarios were selected:

1. Scenario 1 corresponds to the one just discussed, and shown in Figure 4.11.

2. Scenario 2 is based on Scenario 1, with the only difference that the *bst* intervals I_{15} and I_{16} are moved to a separate task, giving four tasks in total.

3. Scenario 3 has a single *gemm mul* task, divided into seven parallel intervals.

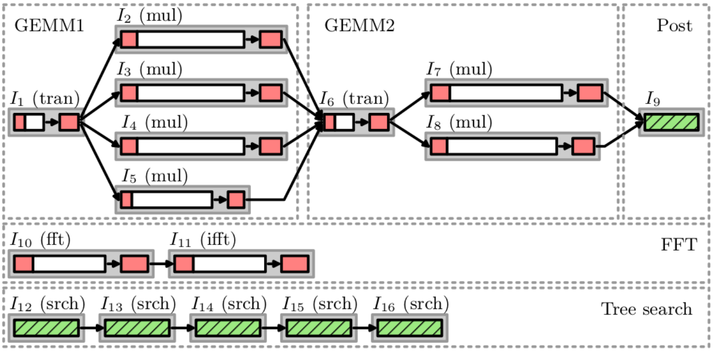


Figure 4.11: A DAG for the intervals of Scenario 1.

4. Scenario 4 has four tasks. First, the same to *gemm* as in Scenario 1, second and third, two independent *fft*, and fourth, only two *bst* intervals.

Additionally, we selected two large scenarios. The first one, Scenario 5, is the sequential composition of eight executions of Scenario 1. This provided a means to compare the heuristic schedulers performance against a known optimal. This comparison is out of scope of this thesis, but details can be found in the PARCO publication on the subject [36]. The second large scenario, Scenario 6, is inspired by real-world applications, with task intervals that may be executed together in practice. In detail, we take inspiration from a KCF tracker [84] (*tracker*), convolutional neural networks (*neural*), control tasks (*control*), and image processing pipelines (*image*).

An overview of the components of this scenario is shown in Figure 4.12. The internal parallelism of the kernels is expressed as a number after the name (e.g. GEMM 4 contains four parallel intervals of *gemm mul*). The first task, inspired by the *tracker*, begins with a a memory intensive task (e.g., opening a file), followed by two parallel chains of convolutions, fast fourier transforms, and matrix multiplications, ending again with a memory intensive interval to simulate writing to disk. The second task, inspired by *neural*, consists of a chain of matrix multiplications to simulate inference on a neural network. The

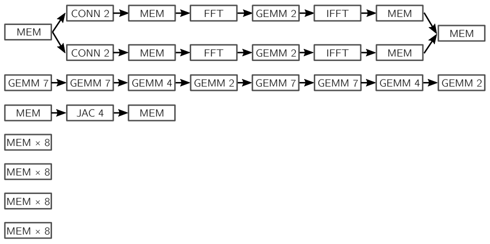


Figure 4.12: An example of a scenario with 110 intervals.

third task is inspired by the *control* application contains the Jacobi kernel, simulating the solving of a system of linear equations. At the bottom there are four tasks inspired by the *image* application, consisting of graph traversal algorithms (here represented by *bst*). Overall, this task combines the components of a possible ADAS-style system, where *image* represents the aquisition of image data, *tracker* and *neural* represent the processing of this data, and lastly *control* represents the actuation on the system.

To generate the schedules for the evaluated system, the execution times of each infividual phase within each interval was measured in isolation on a single core, and the worst-case execution time over 100 executions were selected. Using this information, the PREM scheduling problem was solved using either the ILP [82] or the heuristic [36] schedulers.

4.2.4 System and Experimental Setup

On the TX1 platform, we achieve the required non-preemptive behaviour for PREM systems on CPU [24] by implementing Linux system calls to temporarily enable or disable interrupts on the measurement cores, as well as for flushing and invalidating the entire cache. The latter is required for predictable cache operation, as the *preventive invalidation* approach is only able to clear cache lines belonging to generated PREM intervals – not any cold data left from previous execution. Execution times and cache misses are monitored during the execution of the experiments using the performance monitor's `L2D_CACHE_REFILL` event and the `PMCCNTR` respectively.

We evaluate our PREM compliant scenarios executed according to the solved schedules on 100 000 runs and compare that with an implementation with uncontrolled access to main memory, i.e., *legacy* code. Both implementations are based on a thread pool in order to minimize overheads for creating new threads. Jobs to be executed by the threads are picked from a queue. In PREM execution, the pool has a thread for each CPU core and the queue is ordered according to the schedule generated by the heuristic or ILP solver. When a PREM phase finishes earlier than the WCET, the subsequent phase is executed immediately once all dependencies are satisfied. In *Legacy* executions, the queue is dynamically filled based on the DAG and the jobs are executed by threads whose number equals to the maximum parallelism achievable in the scenario. The threads are scheduled by the Linux `SCHED_FIFO` scheduler and all have the same priority.

4.2.5 Experimental Results

The measured execution times of all 100 000 runs of our small scenarios are presented in logarithmic scale histograms in Figures 4.13a–4.13d. The PREM schedules completion times C_{MAX}, computed by the scheduler based on the *ILP Solver* are shown as a dashed black lines. We calculate the variance P for PREM as

$$P = WCET_{\text{PREM}}/BCET_{\text{PREM}} - 1 \tag{4.5}$$

where $WCET_{\text{PREM}}$ and $BCET_{\text{PREM}}$ are the measured worst and best case execution times of the PREM compliant execution and analogously

$$L = WCET_{\text{Legacy}}/BCET_{\text{Legacy}} - 1 \tag{4.6}$$

for the *Legacy* execution. Finally, we calculate the WCET difference as

$$LP = WCET_{\text{Legacy}}/WCET_{\text{PREM}} - 1 \tag{4.7}$$

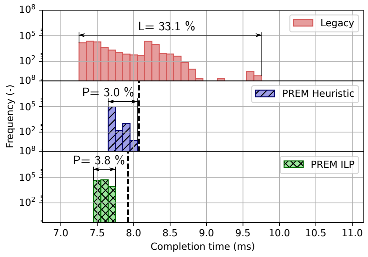
There are two main findings in the results of the experiments. First, in every scenario, the variance of completion times under PREM is small ($P_{max} = 6.1\%$) in comparison to *Legacy* executions (up to $P_{max} = 52.4\%$). The higher variance in *Legacy* executions are caused by a) non-optimal schedules resulting from dynamic scheduling algorithm and b) competition for the shared memory. For example, in Figure 4.13d the *Legacy* execution has three major peaks, corresponding to three different dynamic schedules in which the schedules depends on when each interval finishes[5]. We can clearly see the positive impact of static PREM scheduling in the variance of completion times. The variance could be even smaller if we strictly followed start times of the computed schedule.

Second and most important, the measured WCET of PREM executions is always smaller than the WCET of *Legacy* executions, at least by $LP = 25.1\%$, and up to $LP = 44.7\%$. The WCET of *Legacy* executions is strongly affected by co-scheduling effects. Consider the difference between Scenario 1 and Scenario 2, which only differ in the separation of the memory intensive graph searching task into one or two tasks. In scenario 2, the concurrent execution of intervals I_{12} and I_{15} prolongs both of them up to $3\times$ as can also be seen in Table 4.1. The delay influences the WCET of the *Legacy* execution, which is extended from 9.75 to 11.27 ms. This is prevented by the Predictable Execution Model, as the compiler separates the task into memory and compute phases, and provides the necessary input for the ILP-based scheduler from CTU to produce optimal schedules which prevent memory interference by design. Thus, the execution time of the PREM intervals remain constant.

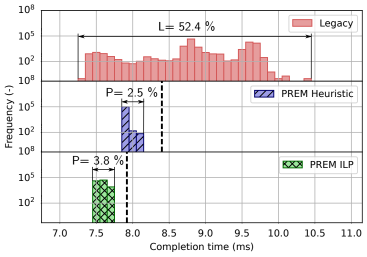
For the large scenarios, where an optimal PREM scheduling solution with the ILP solver was not tractable, the *heuristic* [82] implementation was used instead. The schedule completion times are shown for Scenarios 5 and 6 in Figure and for Scenario 6 in Figure 4.14. The dotted line represents the static schedule completion time found by the heuristic, C_{MAX} as a dotted line.

The average execution time of PREM is higher than that of *Legacy*, which is due to the inclusion of several kernels that were already shown to be less performant under PREM. Even in light of this, PREM provides tighter WCET bounds, $LP = 21.75\,\%$, than *Legacy*. This is a good outcome, as the WCET is the limiting factor in how many tasks that can be successfully scheduled in a system. Also, we can see

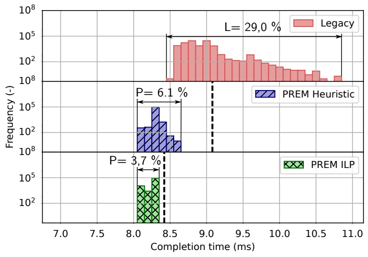
[5]E.g., an earlier completion of an interval could cause the next interval to be migrated from another core instead of taken from the local ready queue.



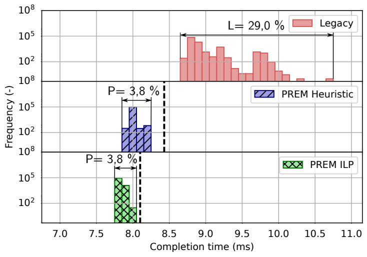
(a) Scenario 1



(b) Scenario 2

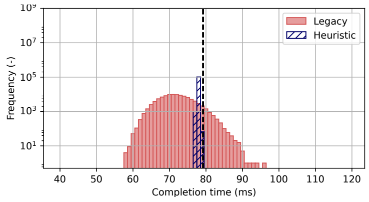


(c) Scenario 3

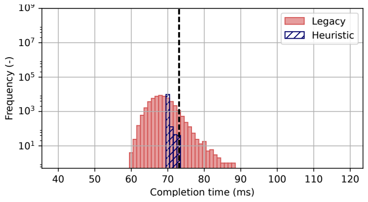


(d) Scenario 4

Figure 4.13: Histograms of completion times of scenarios with and without PREM.



(a) Scenario 5



(b) Scenario 8*1

Figure 4.14: Histograms comparing completion times of scenarios with and without PREM applied

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

Table 4.1: Sample of measured execution times and cache misses for scenarios 1 and 2

that the execution time variance is much lower in the PREM execution ($P_{PREM} = 1.5\,\%$ vs $P_{Legacy} = 65.1\,\%$). From this we see that PREM successfully reduces the execution time jitter, greatly improving the predictability of the system.

Table 4.1 shows the measured execution times and number of cache misses in Scenarios 1 and 2. Each predictable interval has measurements shown for each of the PREM phases (Prefetch, Compute and Write-back). For compatible intervals, the measured values are in the prefetch column only, as compatible intervals only consist of a single memory phase.

From the table two important results can be seen for the memory isolation property of PREM. First, the compute phases of the PREM-compliant executions have a negligible amount of cache misses, even though the cache employs a random replacement policy. This means that even under these conditions, the proposed toolchain is able to produce both a system schedule and transform the code such that the memory isolation property of PREM is upheld in practice. Second, it can be seen that the memory phases of the PREM-compliant executions show an average of 15% fewer cache misses. We believe this is due to the explicit eviction of data that is no longer used, such that the loading of new data is less likely to evict newly loaded data due to the random replacement policy.

4.3 PREM on PULP

Having shown that code compiled with the PREM compiler is able both to provide the necessary freedom from interference guarantees, as well as provide good performance for many benchmarks on the GPU and all tested programs on the CPU, we perform one last evaluation on an additional platform. The contents of this section refers to yet unpublished work done in collaboration with Cyrill Burgener during the completion of his master's thesis [74]. In this case we are looking into the use of hardware-accelerated DMA transfers in the place of SoftDMA memory phases, and the platform for our evaluation is HERO [85] which consists of a soft-core implementation of the Parallel Ultra-Low Power (PULP) [86] accelerator developed at the IIS at ETH Zürich. It consists of a RISC-V based PMCA cluster that is mapped to an FPGA. The cluster consists of 8 in-order *RI5CY* [87] cores with 256 KB of tightly-coupled memory (L1) and an external L2, both implemented as SPMs[6]. The cluster has a DMA capable of streaming data from DRAM to the L1. The cluster is deployed on the programmable logic of the Xilinx Zync-7000 [88], sharing the DRAM with the dual-core ARM A9 host processor. HERO is programmed using OpenMP 4+, and thus remains compatible with the design decisions made for the GPU compiler in Chapter 3.

[6]For the purposes of this evaluation, we only consider the L1 SPM.

Due to its SPM-based Tightly Coupled Data Memory (TCDM) to which all data is manually loaded and stored during execution, the PULP execution model is very similar to PREM, due to the use of DMAs to manually transfer data. This makes these platforms very well suited for PREM: This, in combination with the use of hardware DMA-engines that conform to the principles of SoftDMA (or even were the inspration for it) promise the PREM compiler to deliver two benefits on PULP. First, PREM code should not suffer any of the code transformation overheads that have been discussed in connection to the GPU, and on the CPU. In contrast, the insertion of explicit data movements to and from the TCDM should improve the performance without manually having to manage DMA operations. Second, the use of SPM-based memory hierarchies removes the risk of self-evictions during the PREM compute phase, as has been a significant issue discussed in the two previous sections.

To verify these two benefits, we compile and run benchmarks from the PolyBench-ACC [78] suite on the the HERO [85] Heterogeneous Research Platform. For a realistic setting, we scale the PMCA bandwidth to that of a full-speed (800 MHz) silicon accelerator (vs. 40 MHz on the FPGA)[7]. Section 4.3.2 establishes that the technique achieves freedom from interference, and Section 4.3.1 evaluates the performance of the transformed code.

Furthermore, due to the early stages of development for LLVM support (which is required for the PREM compiler passes) for HERO, there was only a rudimentary OpenMP runtime available. This prevented the evaluation of parallel constructs such as *parallel for*. As such, the evaluation is restricted to a single-core on the PULP cluster executing.

4.3.1 Performance evaluation

We begin with the first promise of the PREM transformations greatly improving performance without having to manually manage data through DMA calls. We do this by measuring the performance of the PREMized benchmarks with hardware accelerated DMA-based memory

[7]Due to this, the CPU bandwidth is considerably smaller than the PMCA's, but in line with previously established numbers [89].

phases inserted by the compiler, and comparing it to the performance of i) the standard OpenMP implementation (i.e., no SPM use), and ii) SoftDMA loops. As the HERO platform is programmed using the OpenMP programming model, the standard OpenMP measurements provides our baseline. The SoftDMA approach is equivalent in access pattern and code generation (except that the inner-most SoftDMA loop is replaced by a call to the 1D DMA-engine).

Figure 4.15 shows that the PREM compiler is able to produce PREM-enabled applications that improve performance over standard OpenMP by up to 11.2×, with an average of 9×. The improvement is possible as OpenMP by design does not consider explicit data movement for SPM memory hierarchies, and therefore every access is handled by DRAM. Through the creation of DMA-enabled PREM memory phases the program is transformed to use the local memory for computation, which explains the improved performance, and delivers on the promise of PREM without overheads.

The use of the hardware accelerated DMA engine also provides a significant speedup compared to the software-based SoftDMA, on average 7.3×, and up to 9.4×. This is because the SoftDMA-style transfers of data using *load* and *store* instructions, which is considerably slower than the DMA engines on in-order processors such as *RI5CY* – motivating the need for a DMA-aware PREM compiler. For most benchmarks, SoftDMA can only improve the performance by a few tens of percent over baseline OpenMP, while for *axpy* performance even decreases. This is because the data copied to the SPM is only used once, and therefore the additional overhead of staging the data through the scratchpad is not amortized over multiple uses. In contrast, even under such conditions the DMA-accelerated memory phases still improves the performance over standard OpenMP by a factor of 4.7×. Conversely, *convolution-2d* shows the maximum speedup for any scheme, as it has the highest level of data reuse.

Following this, we extend our evaluation to also include an evaluation to baseline OpenMP code with manual DMA data management performed by a human expert[8]. While the comparison against baseline OpenMP code provides a means to show the effectiveness of PREM on a platform with similar native execution model, real PULP

[8]The expert is Cyrill Burgener.

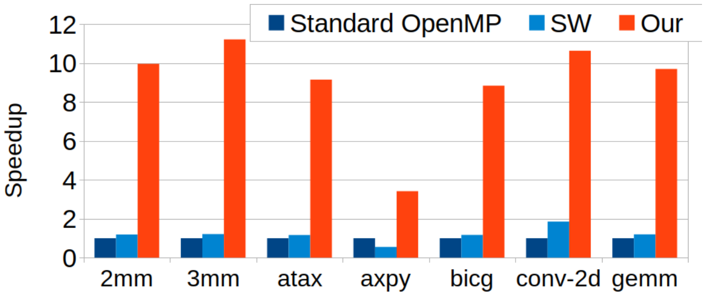


Figure 4.15: Speedup of the code generated by our compiler techniques compared to the standard OpenMP and software load/store instruction data transfers (SW).

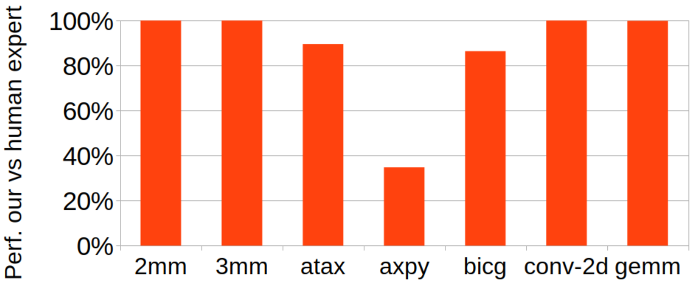


Figure 4.16: The performance of the code generated by the proposed compiler techniques compared to that of a human expert.

applications would indeed have their memory accesses optimixed manually, and as such this provides a more representative comparison in practice.

On average, our technique generates code that reaches 88% of that performance. The poor performance in *axpy* is due to the additional instructions used for tiling, which includes a *signed remainder* operation which has high latency. For *atax* and *bicg* the strict adherence to PREM forces the compiler to generate memory phases that reload some data multiple times, when it is used in multiple intervals. This ensures consistency at the end of each interval, providing the maxi-

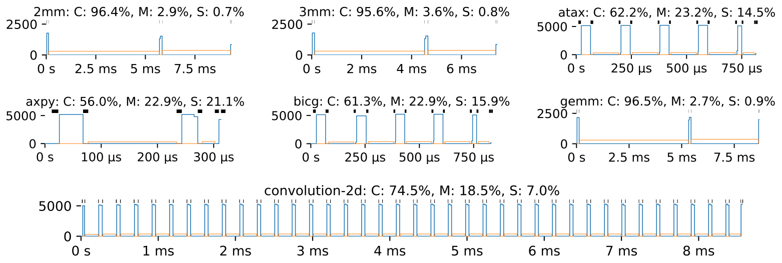


Figure 4.17: The memory bandwidth usage (Y-axis, MB/s) over time for code compiled with the proposed technique, and an interfering CPU task.

mum amount of freedom to a PREM scheduler to schedule intervals from other tasks in between. In contrast, the version optimized by the human expert is only considering performance and thus keeps the data in the SPM for as long as it is needed, and only writes it back to memory after it is used for the last time.

Overall, the performance is similar to that achievable by a human expert, with an additional benefit: This performance is achieved from standard OpenMP code without the need for manual insertion of PULP-specific DMA calls, making the code neater and more portable to other systems.

4.3.2 Host and Accelerator Co-scheduling

Having established that the PREM-compliant code generated by the PREM compiler performs close to what a highly trained programmer can achieve, we now show how the transformed code enables timing-predictable execution. To achieve mutually exclusive memory access between CPU and PMCA, we use the principal ideas of GPUguard, presented in Chapter 2, but with further improvements on the CPU-accelerator co-scheduling developed by Maxim Mattheeuws in his master thesis [90]. We show how this enables completely exclusive access to memory, fulfilling the requirements of PREM.

Similarly to previous experiemental setups we co-execute the PULP benchmarks with a memory intensive PREM-enabled task. This task spinns on memory as soon as it receives the exclusive lock (GPUguard token) for the memory, and stops when it loses it. Our results show the full execution time of the PREM-compliant benchmarks compiled by the PREM compiler, with benchmarks using only 1/8th of the local SPM capacity. The reason for this is that, due to the limited LLVM support, this allows us to emulate a perfect linear speedup to 8 cores by compressing the execution time of the compute phases on PULP by a factor of 1/8th. This had not been possible if the single core had used the entire capacity. While not optimal, this case is interesting as it represents the worst case scenario for how large percentage of the time that the DMA is holding the exclusive access to the memory, and limiting memory accesses from the CPU. As the DMA operation is done in hardware, we do not scale this in the presented plots.

Under these conditions, Figure 4.17 shows the memory bandwidth utilization by PULP (blue lines) and the CPU (red lines) throughout the execution of each benchmark. Each plot is labeled with the percentage of time that is spent in the PREM *compute* (C) and *memory* (M) phases, as well as the time spent on synchronization (S), which are shown as black lines at the top of the plots.

The most important finding is that for every benchmark, the memory accesses are completely separated in time, showing that the PREM promise of freedom from interference is achieved: CPU tasks are never affected by the bandwidth use of PULP, or vice versa.

Another important finding is that a significant portion of the *memory time* remains free for the CPU to use during the PULP *compute* (C) phases. On average 77.5% of the time is available for the CPU to access memory, and at least 56%. This means that, finding a feasible schedule for CPU tasks is not significantly complicated when using a PULP in a PREM system, even under the worst-case linear speedup to the compute phases.

Lastly, the *synchronization cost* (S) required to enforce the mutual exclusion adds on average 8.7% to the total execution time. The highest cost is recorded for *axpy*, which is a notoriously light-weight kernel and as such has very short PREM phases. Even in such a unfavorable case the synchronization time is only 21.1% on PULP.

4.4 Conclusion

In this chapter we have evaluated the PREM compiler presented in Chapter 3 on three different platforms. We have seen how the various code transformation implemented affect the performance of the resulting PREM code. In particular, we have seen that on the GPU, previously presented techniques for generating memory-accessing phases such as Decoupled Access Execute (DAE) – generally referred to in this thesis as *stripping techniques* do not perform well on the GPU. Instead, we have shown that thanks to the static analysis that the PREM compiler does for footprint calculations, better memory phases can be generated through the technique we call SoftDMA. Furthermore, we have seen that similar code generation techniques such as SoftDMA allows for the generation of slim and well-performing memory phases also on cache-based CPU systems, leading to well-performing PREM intervals.

Overall, we have shown that a key aspect in how well PREM performs is the ability of the compiler to move execution time from the, at a system level, mutually exclusive memory phases T_{memory} to the compute phases $T_{compute}$, which are freely schedulable without taking into account global resource (memory) utilization. The SoftDMA scheme is very-well suited for this, in comparison to other techniques presented in the literature.

Additionally, we have seen that the PREM synchronization cost, especially in the context of heterogeneous execution, can severely impact the performance. On the GPU we saw that kernels with low temporal data locality require frequent refills of the local PREM data buffers, which implies frequent synchronizations and large overheads. We have shown that this limitation comes from the close interplay between the size of the local memory, the cost of synchronization, and the maximum synchronization frequency. In contrast to the GPU evaluation, where this was a problem already for slightly memory bound kernels, we saw that the relatively larger local memories on the PULP platform addressed this problem by making the synchronization costs negligible.

Importantly, we have seen that on every evaluated platform, the compiler has been able to create PREM intervals that are schedulable on each system without leading to global memory accesses in the compute phases. Resulting from this, we have seen that the PREMized systems have a near-zero impact on their execution times in isolation when significant external memory interference is introduced in the system. This shows that the proposed PREM techniques are indeed able to provide the promised freedom-from-interference guarantees of PREM.

Overall, we have seen that SPM-based systems are, as expected, better suited for PREM due to their software managed nature. In the experiment we executed on a cache-based system, we saw that the hardware-managed replacement policy of the cache required special consideration to achieve good results with PREM. Furthermore, we saw on the NVIDIA GPU that the small size of the SPM lead to significant synchronization overheads, which even in light of this make it interesting to explore the use of the much larger GPU caches as local storage.

From this, it is clear that further investigation is required on the use of caches in PREM, which will be the focus of the next chapter.

Chapter 5

Taming Data Caches

So far we have seen that the PREM compiler, in collaboration with runtime techniques such as GPUguard, as well as external PREM schedulers is able to produce programs that remain predictable also under external interference. However, there have also been issues related to the behavior of the local storage used for PREM.

This chapter addresses two such questions raised in the previous chapters. We begin in Section 5.1 by addressing the question of the synchronization wall issues with PREM execution in CPU-GPU systems. In our experimental results we saw that several kernels were not able to ammortize the synchronization overhead due to frequent refills of the small SPM. In this chapter, we explore the solving of this issue by using the much larger last level caches for local storage.

Following this, in Section 5.2, we address the issue with the last level cache of the CPU subsystem, identified in Section 4.2. In that setup we had to make several restrictions to allow multiple PREM tasks to execute in parallel on the shared data caches. In this section, we look into these aspects in more detail, and propose more robust alternatives.

5.1 Using Large Data Caches to Reduce Overheads for Heterogeneous PREM

Beginning with the GPU, this section presents work that was published at *DATE'19* [38], addressing the issue of high synchronization overheads for certain kernels in heterogeneous PREM environments. PREM synchronization to coordinate memory accesses between CPU and GPU, using techniques such as GPUguard, introduces an overhead in the program execution. In Section 4.1.2 we saw that the size of the local memory plays an important role in this overhead, as more frequent, and fixed-length synchronizations are less well amortized over PREM phases of smaller length. On the GPU the small SPM was cause for significant overheads in several benchmarks. Furthermore, due to the minimum granularity at which the synchronizations can be performed, due to a minimal inter-arrival time of timer interrupts used to trigger the synchronization, further overheads were introduced in several benchmarks. We refered to the later form of overhead as hitting the *synchronization wall.*

Let us begin by zooming in on the problem to establish where the problem occurs, and how it is triggered. When executing PREM workloads that encompass both CPU and GPU exceutions, the scope of the PREM scheduler is extended from the management of processor time within the CPU (i.e., as for a common OS scheduler) to also schedule access to global memory. To incorporate the GPU in this scheme, we proposed GPUguard in Chapter 2 to pass memory tokens between CPU and GPU based on schedule events. These synchronizations events are triggered by timer interrupts at the expiry of a WCET watchdog timer (Figure 5.1 (a)), at which point the memory *token* can be exchanged between the processing units (Fig. 5.1 (b)). However, these synchronizations can not occur at too fine granularity: The system requires enough time to accomodate the interrupt latency and interrupt handler, as well as leaving enough time for "useful work". Therefore, there exists a system-dependent *minimum synchronization granularity* (MSG) (Figure 5.1 (c)). In the case when the phase lengths are shorter than the MSG, the GPU kernel is forced to *idle* until the CPU becomes ready for synchronization (Figure 5.1 (d)). This corresponds to the kernel hitting the synchronization wall,

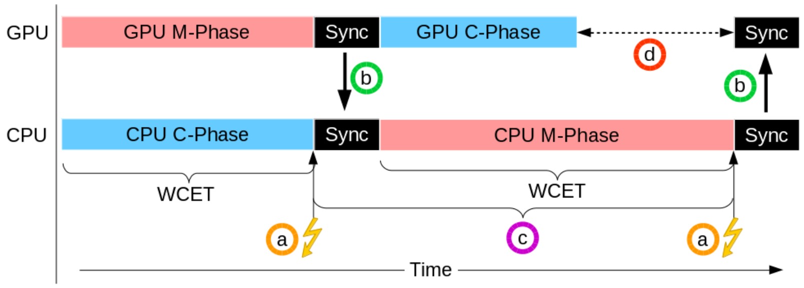


Figure 5.1: The key components of a PREM interval.

as discussed in Section 4.1.2. When designing the system to account for the MSG and the length of the PREM phases, we say that we place a *budget* on the system, which affects when the watchdog timer expires. For the GPU, these budgets are translated into GPUguard scheduling quanta per the discussion in Section 2.1.2.

The impact of these effects is closely connected to the size of the local storage used for PREM, following from Equation 2.1. Smaller local memories mean that the data must be exchanged more often as the computation progresses. For each refill the memory token must be requested through the GPUguard synchronization, and the corresponding synchronization overhead paid. Additionally, the synchronization may be further delayed if the MSG comes into play. For larger local memories, the local memory can be refilled less often, leading to fewer such synchronizations overall, and a decreased likelihood that the phase lenghts are short enough to trigger delays due to the MSG.

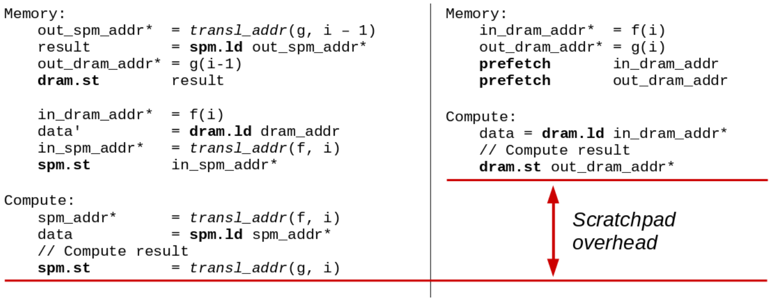
For the GPU, the SPM presents an attractive candidate for local storage in PREM, due to its predictable behavior, achieved by being completely software controlled. However in the NVIDIA GPUs, the SPM is limited to only 48KB of storage, which causes frequent synchronizations, and gives raise to the problems identified above. To overcome these problems, this chapter investigates how the larger hardware-managed caches of the NVIDIA Tegra SoCs [41] can be successfully used with the PREM model to achieve predictable execution at better performance than the SPM-based state-of-the-art.

5.1.1 The Cases For and Against Caches

This section describes the fundamental differences with the use of SPM or caches on the NVIDIA GPUs. The first point summarizes the fundamental reason why caches are of interest in the scope of the findings of this thesis, but continues to outline additional differences that impact PREM execution on the GPU and that will be further explored.

Synchronization The main issue with issue we are trying to solve is that the small size of the SPM implies short PREM phases, even below the MSG, which causes the synchronization/idling overhead to blow up. Intuitively, these overheads can be overcome by increasing the granulary of the *intervals*, such that the synchronization makes up a smaller proportion of the overall execution time. On current generation heterogeneous SoCs, the last level cache (LLC) of the accelerator is much larger than the SPM ($5\times$ on the NVIDIA TX1, $10\times$ on the TX2). Thus, the use of caches promises a more effective use of the kernel execution time.

Code performance As SPMs are software managed and explicitly addressed, they require a significant addition of instructions to manage the data allocation and data movement. This implies an overhead compared to the use of implicitly addressed caches. For some kernels, such as matrix multiplication, the amount of data reuse makes it worth the additional instructions to bring the data closer to the cores. However, for many kernels , i.e., the same type of kernels that suffer from synchronization overhead, this overhead can be significant. A simple example provided in Figure 5.2 highlights the difference between the two cases. Note that depending on the complexity of the address calculation, the added instructions from `transl_addr` (which transforms a DRAM address to its SPM counterpart) can be significant. In contrast, the only instructions needed for hardware-managed caches is a prefetch of the original address in the memory phase, as shown in Figure 5.2. Because of this, hardware caches promise additional performance benefits due to hardware-managed data placement.



```
Memory:
    out_spm_addr*   = transl_addr(g, i - 1)
    result          = spm.ld out_spm_addr*
    out_dram_addr* = g(i-1)
    dram.st         result

    in_dram_addr*   = f(i)
    data'           = dram.ld dram_addr
    in_spm_addr*    = transl_addr(f, i)
    spm.st          in_spm_addr*

Compute:
    spm_addr*       = transl_addr(f, i)
    data            = spm.ld spm_addr*
    // Compute result
    spm.st          = transl_addr(g, i)
```

```
Memory:
    in_dram_addr*   = f(i)
    out_dram_addr* = g(i)
    prefetch        in_dram_addr
    prefetch        out_dram_addr

Compute:
    data = dram.ld in_dram_addr*
    // Compute result
    dram.st out_dram_addr*
```

Figure 5.2: SPM data movement code (left) requires more instructions than caches (right).

Self-eviction To ensure predictability, no cache miss must occur in the compute phase. To measure this we use the *compute phase miss rate* (CPMR), defined as the ratio of cache misses in the compute phase over the total amount of cache misses. A CPMR of zero thus means that all cache misses occur in the memory, which in turn enables predictable execution.

Thus, the key design goal for PREM is to minimize the CPMR, which makes a strong case for the SPM: The software managed data movement ensures that data brought into the SPM is guaranteed to *survive* until explicitly *evicted*. In contrast, data in the caches are managed by a fixed *replacement policy*, that selects which data to *evict* when new data is requested. If the memory accesses of a program causes the replacement policy to evict *live data*, the program is subject to *self-eviction*. In the case of the *least-recently-used* (LRU) [35] replacement policy this represents no problem, as stale data will always be evicted before data that was part of the latest prefetch phase. However, LRU is seldom used in commercial systems, because of the complex hardware needed to implement it. Instead, cheaper but less predictable replacement policies are used, that can lead to (more or less) random *self-evictions* of alive data from the cache. This is also the case in the NVIDIA Tegra GPUs [81].

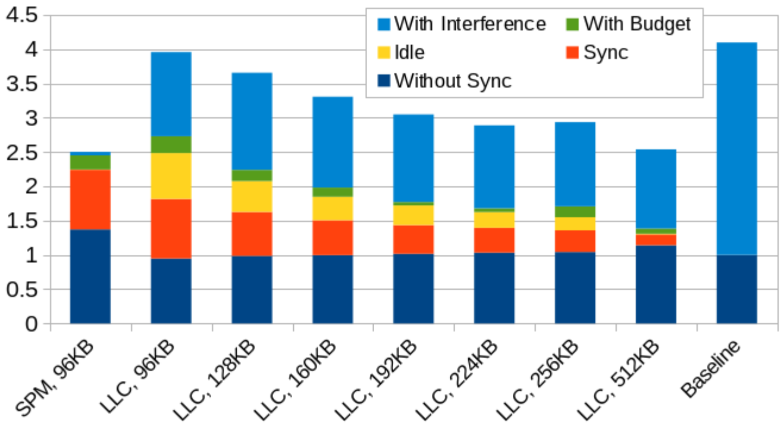


Figure 5.3: The breakdown of the execution time for the *bicg-100* kernel on the SPM, LLC, and without PREM (baseline).

5.1.2 SPM and Cache Differences in Practice

In summmary, the cache promises smaller overheads, but the problem of self-eviction may compromise the key design goal of minimizing the CPMR. To see how this manifests in practice, we run the cache-friendly *bicg-100* kernel from the PolyBench-ACC [78] benchmark suite on the NVIDIA TX1. This kernel is interesting for two reasons. First, it is one of the kernels identified in Section 4.1.2 to suffer from the overheads due to the SPM begin too small, leading to large overheads. Second, this kernel has cache friendly access patterns, which should limit the impact of self-evictions, while still allowing to demonstrate the benefits of larger local storage, and the disappearance of the synchronization overheads.

We compare the characteristics of this kernel on the SPM, on the LLC, and without PREM (baseline), under different interval sizes T, determined by the amount of data touched. The SPM is limited to $T \leq 2{\times}48KB$ (kernel is distributed over two clusters, each with access to their own SPM), while the LLC allows $T \leq 256KB$. The results are shown in Figure 5.3, where execution times are shown relative to the baseline, i.e., the original program without PREM transformations or synchronizations. The bars are broken down to highlight different effects.

From the bottom up, “without sync” shows the effect of either SPM data movements or prefetches as created by the PREM compiler, and the “idle” and “sync” parts show the two sources of synchronization overhead described in Section 5.1.1. In these aspects, caches indeed do better than the SPM; the overheads are rapidly decreasing as T goes up, and the use of prefetches initially has a positive effect, as can be seen in the decrease in “without sync” between the SPM and LLC cases. As T increases beyond the cache size of $256KB$, we start seeing capacity misses, as illustrated by an increase in the “without sync” bar at $512KB$.

After budgeting for the WCET of the intervals and applying memory interference, the two top-most bars show a large difference between the SPM and the LLC. Here, the “with budget” part of the bar corresponds to the difference between the budgeted WCET and the actual execution times of the phases, as given by I_{budget} in Section 4.1.1. The “with interference” part of the bar indicates how much the execution time increases under external memory interference, achieved in the same manner as in Section 4.1. In a correct PREM system, this value should be negligible, as all data is stored locally and not subject to external interference. This result is indeed achieved on the SPM. When executing on the cache however, a significant slowdown is visible under memory interfere. Compared to the baseline, PREM on LLC can still perform better under interference, because cache misses that occur in the memory phase are protected by GPUguard. However, cache misses in the compute phase (i.e., violation of PREM) are subject to the same slowdown as in the baseline case.

For the LLC, the bad performance under interference would occur only if the data prefetched in the memory phase is selected for eviction before its point of use in the compute phase. The compiler-generated prefetch phases are guaranteed to issue prefetch requests for the data, which implies that the cache replacement policy is working against us, as the prefetch can not guarantee that the data is in the cache after the memory phase.

5.1.3 The Way of the Cache

The main source of self-evictions in the NVIDIA GPU caches, as shown by [81], is the random replacement policy. Data stored in different cache ways are more or less likely to be evicted, and if the loaded data ends up in a cache way that is more likely to get evicted, it will not *survive* until the start of the compute phase. The authors of [81] were only able to show this effect in the L1 cache, where out of four *cache ways* per set, one was three times more likely to evict the data[1]. As new accessed data will be stored into the set where the previous data was evicted from, each new accessed datum has a 50% chance of being stored to the fourth way. We refer to this way as the *bad way*, because data there is much more likely to be evicted on the next cache miss. Correspondingly, we refer to the remaining as the *good ways*. To minimize the CPMR we want to have a near-zero probability that data is stored in the *bad way*. Since the eviction probabilities imply a 50% chance of data being stored to the *bad way*, we can model this as a *coin toss* (probability of getting R *heads* in a row), in which the likelyhood of using a *bad way* reaches a probability of less than 0.5% at $R \geq 8$.

We verify this intuition, and that this applies to the LLC, in Figure 5.4, measuring the CPMR for different R at interval sizes T. As expected, increasing the number of prefetches by a factor of R monotonically decreases the CPMR towards near-zero values. Thus, we can decrease the CPMR by choosing $R = 8$, and we refer to R as the prefetch repetition factor.

In the other dimension we explore the effects on the CPMR as the interval size T increases, and see that as T decreases the CPMR also decreases. For all $T \leq 192$ the CPMR reaches CPMR $< 10\%$, after which it increases rapidly. We know that 3/4 of the cache ways are *good*, and 1/4 is *bad*. That also means, that of the full cache capacity of $256KB$, only $192KB$ (3/4th) is available in *good* cache ways. Which in turn means:

- While $T \leq 192KB$, all data fits in the *good ways*, and with a high enough R, all data will reside there.

[1]In [81], the observed probabilities of evictions were $(\frac{1}{6}, \frac{1}{6}, \frac{3}{6}, \frac{1}{6})$.

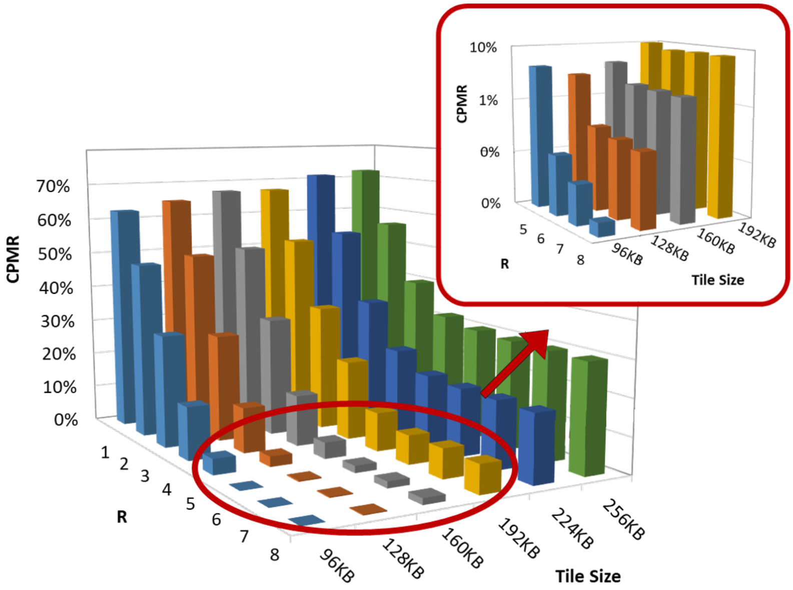


Figure 5.4: The CPMR for different R and T.

- Once $T > 192KB$, data must also be stored in the *bad ways*, and there is a very high risk that this data will be evicted at any cache miss. We therefore expect the CPMR to start increasing due to self-eviction.

Comparing this to the CPMR in Figure 5.4, we can see that this matches the observed pattern. Thus, by choosing an interval size T that is small enough to fit in the *good ways* of the cache, and repeating the prefetch operation $R = 8$ times, we are able to significantly reduce the CPMR, whose previously high values prevented predictable execution on the LLC.

Example Revisited

With this information, it is possible to reduce the amount of cache misses by repeating each prefetch operation $R = 8$ times. The results for the *bicg-100* kernel shown before are presented in Figure 5.5.

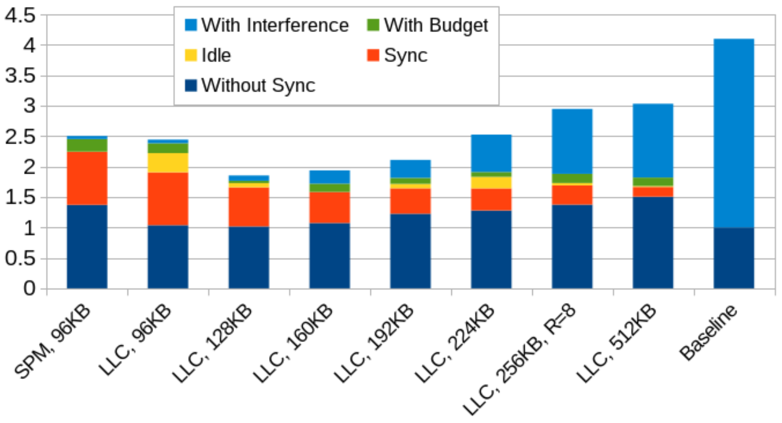


Figure 5.5: The breakdown of the execution time for the *bicg-100* kernel, with a prefetch repetition R of 8.

In this configuration the good properties of reduced synchronization overhead is preserved, but the effect on code overhead and interference has changed for the negative and positive respectively. Beginning with the code overhead, which now includes repeated prefetches (and is shown in the "without sync" part of the bars), we can see that with $R = 8$ it slowly increases with the tile size T. The reason for this is visible in Figure 5.4: As the tile size becomes larger, the CPMR hits a plateau at a higher and higher value, meaning that as T gets larger we can never reach the case where every memory access in the compute is a cache hit. To understand how this affects the execution time, lets reason about the impact of repeated prefetches. For a repeated prefetch that hits in the cache, the increase in execution time should be negligible because the low cache latency. If a repeated prefetch causes a miss, we will overcome the self-eviction by refetching it within the memory phase, and thus move one cache miss from the compute phase to the memory phase, expecting no change in the overall execution time. This is the case where the CPMR reaches near-zero values, for tile sizes that fit in the *good cache ways*, when $T \leq 192KB$. However, if we repeat the prefetch and miss multiple

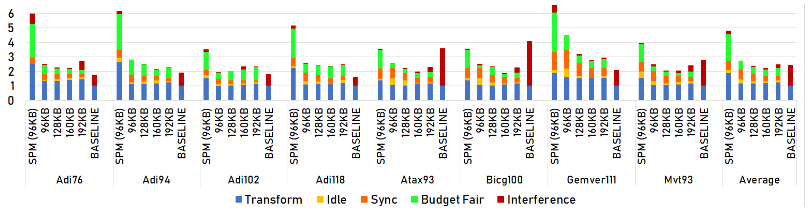


Figure 5.6: The results for the individual kernels in *fair* co-scheduling with the CPU.

times in the memory phase, and after that *still* miss in the compute phase, the overall execution time goes up because the increased overall number of cache misses. We can see that this effect starts at $T = 192KB$, where the *good* cache ways are no longer enough to hold all data touched by the tile.

The good news, however, is that the lowering of the CPMR has the corresponding decrease in sensitivity to memory interference, showing a positive relationship between the CPMR and predictability achievable.

5.1.4 Evaluation

This section extends the evaluation to more kernels from the PolyBench-ACC benchmark suite [78], using a subset of benchmarks for which the SPM-based PREM execution implies large overheads. The experiments are executed on the NVIDIA Tegra TX1 [41], using the same approach as presented in Section 4.1.1. As we have seen that the CPMR for intervals larger than $192KB$, i.e., larger than the size of the good sets, is significantly increased, we focus the evaluation on smaller intervals that are less susceptible to interference. The results are presented in Figure 5.6.

For each kernel we measure the execution time without GPUguard to determine the effects of the cache-based PREM code transformation done by the compiler, in comparison to the SPM and to an unmodified baseline ("Transform"). We also measure the additional overhead due to synchronization and idleness introduced into the system, as well as the budgeting, as described in Section 4.1.1, to account for the WCET.

For a CPMR of zero, the WCET in isolation and under interference will be the same, and freedom from interference achieved.

We co-schedule the TX1 CPU and GPU so that both devices get an equal share of the memory bandwidth, which ensures that neither device is starved for memory. This is achieved by budgeting the memory and compute phases to equal length. The results are presented in Figure 5.6.

Optimized cache performance In all cases, the SPM-based state-of-the-art performs significantly worse than the LLC, and we can confirm the previous results that the best configuration is to use a T that only depends on the *good cache ways*. For the best interval size $T = 160KB$ the LLC performs, on average, twice as good as the SPM, due to the coarser granularity of synchronization made possible by the large LLC. An additional effect of the cache-based approach is that the length of the memory and compute phases become more balanced, leading to smaller amount of idleness budgeted into the system schedule. In the case of SPMs, the compute phase was so short that it would starve the CPU from using memory, but on the LLC, the longer latency to the LLC brings up the execution time, allowing the CPU to get a fair share of memory. The increase in compute phase latency is compensated by the lower complexity of the LLC memory phase (Figure 5.2).

We also see that on average, the LLC outperforms the baseline under interference. While the average over these kernels is modest, only 10% improvement, the SPM on average showed a almost a 200% *decrease* in performance, even under interference. However, in the best case, PREM on LLC offers a 215% improvement in WCET compared to the baseline. As the interference to the baseline kernels are found by measurement, they are only a lower bound on the possible slowdown. In contrast, PREM is designed with this limit as a design goal.

Predictability In addition to better performance, the predictability guarantees can still be preserved with the LLC. Figure 5.7 shows how much, on average, the execution time increases under interference. For intervals similar in size to the SPM, $T \leq 128KB$, the interference only adds 3% to the execution time. For the $160KB$ interval the

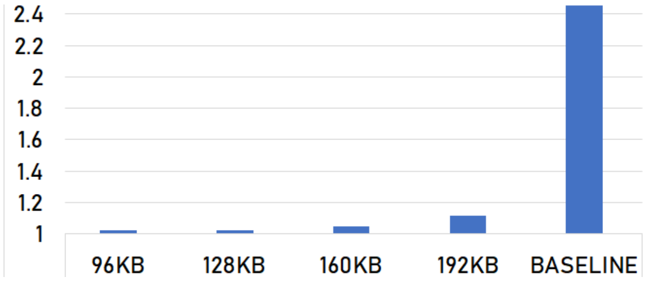


Figure 5.7: The average sensitivity to interference for all kernels.

sensitivity increases to 5% over the execution time in isolation, and for 192KB, at the limit of the size of the *good ways*, the sensitivity increases further to 15%. However, this is significantly less than the 245% for the unmodified baseline.

Thus, it has been shown that with larger local memories, the synchronization overhead inherent to heterogeneous PREM can be successfully managed also for low-complexity kernels, by selecting a platform with larger local memories. In addition to this, it has been shown that, while perhaps not suitable for modeling and static analysis necessary for safety certification, mechanisms within GPU caches can be used to ensure better cache hit ratios after prefetching.

5.2 Managing Random Replacement Policies and Shared Cache Levels in CPU Caches

Let us now look a bit closer into the behavior of the CPU caches as well. Following the discussion in Section 4.2.1, in the submitted paper to TECS [69] we further evaluate the effects of NVIDIA Jetson TX2 caches on PREM intervals. While the experimental results from this evaluation comes from the newer TX2 platform, the concepts are applicable also to the previous-generation TX1 used for the evaluation in the previous sections. The main difference between the TX1 and the TX2 is that the TX2 features a newer generation of GPU as well as, besides the A57's also in the TX1, an additional two-core processor of NVIDIA's own Carmel make. However, for this evaluation we only consider the A57s.

As outlined in Section 3.2.2, the footprint analysis internal to the compiler is able to account for cache line locality, but it does not account for set conflicts in the resulting PREM intervals. The main goal of this section is therefore to evaluate the impact that this has on the resulting PREM intervals and how well they manage to isolate the execution phase from use of the shared memory system.

5.2.1 Cache Associativity, Size, and Misses

With preventive invalidation, as outlined in Section 4.2.1, set-conflicts can only occur *within* an interval, as long as private caches are used. The TX2 (and TX1) only offers a single core-private memory, which is the L1 cache. Therefore, this is the obvious choice for local storage for PREM intervals on the TX2.

To test the impact of set-conflicts on the PREM intervals, we therefore use a set of PolyBench-ACC [73] benchmarks and configure the compiler to create intervals smaller than the 32KB L1 cache, execute them, and record the number of set-conflicts in Figure 5.8. For many benchmarks, over 20% of the prefetches cause a set-conflict, i.e., extending the previous example, a new datum d_2^{new} replaces either d_0^{new} or d_1^{new} within the interval itself – again violating the PREM isolation guarantees by causing a compute phase miss. In these benchmarks it is caused by several 1- and 2-dimensional arrays having overlapping accesses to the same sets, which quickly increases beyond the 2-associativity of the L1 cache.

We therefore conclude that for otherwise correct PREM intervals, their execution in practice violates the PREM isolation guarantees due to frequent self-evictions due to set-conflicts in the low-associativity L1 caches. This at first glance is a discouraging find. However, if we can apply some cache partitioning technique to the shared L2 cache, we would be able to isolate different tasks from interfering with each other, which would make the L2 cache an option for local storage for PREM.

The L2 cache has 8× higher associativity and number of sets than the L1, and can be expected to perform better for these benchmarks.

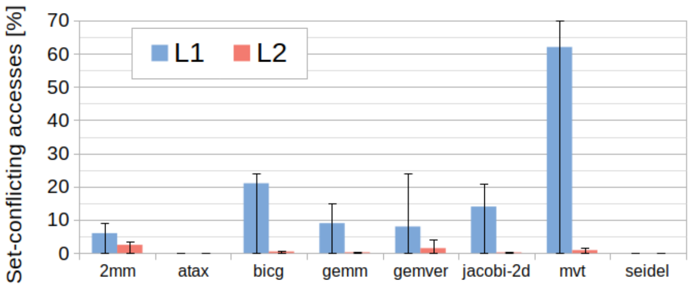


Figure 5.8: Average (and min/max) set-conflicts *per interval*, for L1 and L2 caches.

We will address the shared status of the L2 shortly, but first let us establish that the L2 solves the problem of self-eviction due to set conflicts. The benchmarks are recompiled with a target interval size of 2 MB, ensuring that the full L2 cache is used. Figure 5.8 shows that the number of set-conflicts per interval are now always below 5%, and typically below 2% – a 10× improvement compared to the L1. We conclude that the larger amount of sets and higher associativity makes the L2 less susceptible to set-conflicts for these benchmarks.

Set-conflicts could potentially be avoided in compiler-generated PREM intervals if the set indexing bits of each accessed data could be determined. This would allow the compiler to avoid the creation of intervals in which one or more sets are over-occupied.

For structured data types, e.g., arrays and structs, an approximation of the set pressure could be generated by assuming an arbitrary address a_{guess} for the beginning of the structure, and tracking how the set indexing bits change for each element within the structure. This is possible as the offset of each element in an array or struct is known, and the indexing bits for each element follow from the initial guess a_{guess}. For example, the indexing bits could be extracted for element n in an array of type T as $a_n = a_{guess} + n * sizeof(T)\ \&\ SET\_INDEX\_MASK$, where SET_INDEX_MASK is the bitmask of the indexing bits and & the bit-wise *and* operator. An incorrect guess for a_{guess} would affect which sets that were used, but not their set pressure. However, it is not possible to derive the relative location of two different data structures A and B at compile time (this depends on the linker, heap allocators, and stack layout), which would prevent such an approximation to work between data structures. This, as a correct approximation of set pressure requires the assumed starting address of the second structure a^B_{guess} to be correct relative to the first a^A_{guess}.

Potentially, this problem could be solved by enforcing specific memory layouts of data structures by linker scripts, or by collecting all data structures within a *container structure* C – e.g., a struct which has the other data structures (e.g., A and B) as members. From a single initial guess for the start of the container a^C_{guess} the addresses $a^A = a^C_{guess}$ and $a^B = a^C_{guess} + sizeof(A)$ would follow, and the indexing bits could be extracted. However, as there are several problems to to implement such a scheme for data present in static memory, the heap, and the stack, such explorations are out of scope of this work, and we rely on this post-compilation validation of the cache behavior.

We highlight that set-conflicts in PREM compute phases are comparable to non-contended single-cores with dedicated caches, easily upper-bounded using existing single-core timing analyses. Since memory phases of other tasks are contended only by set-conflicts of co-executing compute phases, and the compute phase is inflated only by the contended set-conflicts, the resulting WCET of PREM tasks are significantly smaller than non-PREM. While we strive for zero conflict misses, we accept the three benchmarks that still experience a few percent of set-conflict misses.

Following this, we select to use the TX2 L2 cache for local storage for our PREM intervals. In the next section we will introduce cache coloring to ensure that the isolation properties are upheld also between tasks in the context of a shared cache. First, let us evaluate the cost of preventive invalidation, the orthogonal technique that we use to ensure that there are no self-evictions within the task itself. From Section 4.2 we know that, besides the need to verify set-conflicts post-compilation, the preventive invalidation required to ensure predictable cache line replacement in the A57 caches adds an overhead. As part of our evaluation in the TECS submission [69] we measured the overheads of the preventive invalidation approach on the benchmarks used in this evaluation.

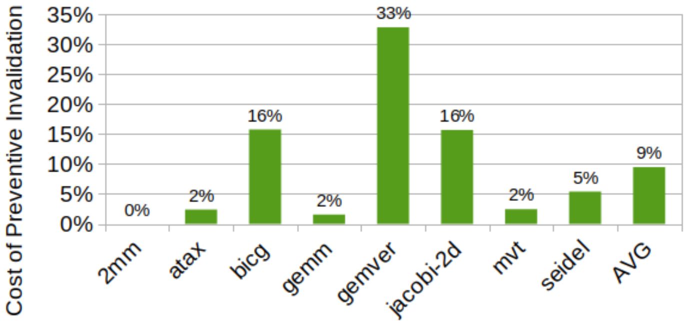


Figure 5.9: The slowdown factor of preventive invalidation.

Figure 5.9 shows that *preventive invalidation* on average adds 9% to the execution time, due to additional cache refills. Note that this is lower than presented in the previous section: As part of the ongoing compiler development we identified that the preventive invalidation routine did not account for cache line locality, as the SoftDMA prefetch phases would do. Importantly, this did not lead to incorrect results, as the cache line was evicted multiple times instead of just once. Thus we were able to optimize this stage by ensuring that each individual cache line was only evicted once, without loss of correctness. From this the overhead dropped significantly, corresponding to the same type of speedup as presented for cache-based SoftDMA in Section 3.2.3.

Note that data is restored to the cache during the next prefetch phase, ensuring correct PREM operation. As outlined, invalidation is necessary for PREM with RRP. For predictable policies, e.g., Least Recently Used (LRU), writeback phase invalidations are not necessary, removing this overhead. However, the performance is acceptable, as we are willing to sacrifice some of the peak performance to achieve predictability.

5.2.2 Cache Coloring, Shared Caches, and PREM

Shared caches mean that the impact of L2 interference between tasks on different cores needs to be considered. This can be addressed with *cache coloring*, which ensures isolation of cache lines between cores by assigning a fixed number of cache sets to each core. Note that cache coloring does not inherently prevent self-eviction within a task (a problem we solved with *preventive invalidation*), but only ensures that two *separate* tasks can not interfere with each other. Thus, cache coloring is an orthogonal approach to PREM, which we combine to ensure full L2 isolation. As has been shown by Kloda et al [37], this can be achieved transparently to applications, by managing it through the virtual memory system. We collaborated with the team behind [37] at the University of Modena and Reggio Emilia to employ this technique to ensure PREM isolation also in the context of a shared L2 cache.

Based on their approach, we use the Jailhouse hypervisor [91] that implements application-transparent cache colored [37, 92] *inmates* (VMs). We execute all benchmarks as bare-metal inmates, together with a PREM-compatible synthetic benchmark that generates the maximum amount of interference possible[2]. The hypervisor is extended to implement *memory mutices*, or *memtex* for short, which are requested by hypercalls implemented in the `__prem_notify()` runtime functions inserted as scheduler hooks by the PREM compiler. Only a single task can hold the memtex at once, ensuring complete isolation to main memory, as per the PREM definition.

This presents a realistic setup of a PREM system, and our evaluation has two goals. First, we need to ensure that the cache coloring support provided by the hypervisor ensures that the PREM tasks do not encounter interference from other tasks that evict their cache lines during the compute phase. Second, we want to evaluate the performance of the benchmarks on this setup, compared to running them as standard applications without cache coloring or PREM transformations and scheduling.

Cache Line Isolation A task is correct under PREM if all shared memory accesses (cache misses) occur in the memory phase. Here we will establish that this is the case for the compiled code. Since the L2 cache is shared, the interval sizes are reduced from 2 MB to 512 KB, i.e., one quarter per A57 core, using cache coloring to ensure

[2]By spinning over an array at the stride of the length of a cache line.

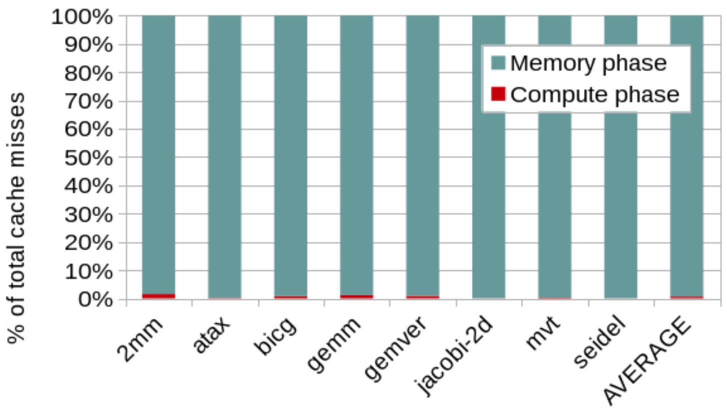


Figure 5.10: Cache misses in the memory and compute phases.

isolation. We use the A57 performance monitoring unit (PMU) to record all cache misses occurring in the compute and memory phases separately, presenting the results in Figure 5.10.

We confirm that on average over all the benchmarks, 99.4% of the cache misses occur in the memory phase. In the worst case, for *2mm*, the number is 98.5%. The few remaining misses can be traced back to the set-conflicts shown in Section 5.2.1, and to *instruction* misses (also handled by the L2), as the presented techniques only account for *data* accesses, as outlined in Section 3.3.

Importantly, for benchmarks were the move from L1 to L2 could remove all set-conflicts, the number of cache misses in the compute phase remains very low, highlighting the benefits and possibilities of PREM in combination with cache coloring in combination with larger caches of higher associativity.

Regarding instruction misses, these could be prefetched as part of the generated PREM prefetch phase. As the PREM compiler outlines all PREM phases into separate functions, the prefetch phase could use the pointer to the compute function to prefetch its data. However, this would require further changes to the compiler backend as the length of the function is not known until the machine code has been generated, information that is required to know the length of the code to prefetch.

The much more difficult problem of prefetching data, which requires significant compiler analysis to determine memory footprint and select PREM intervals removes the vast majority of misses, and PREM isolation can be effectively provided. Cache coloring guarantees that the number of cache misses will not change due to the activity of other cores, ensuring that full isolation is achieved and cache miss analysis can be performed with single-core methods. With this, we can show that our PREM solution can be used successfully on random replacement caches that are common in practice.

Performance Evaluation This section evaluates the performance of the PREM tasks on the NVIDIA TX2 with hypervisor-based cache coloring and PREM enforcement through *memtex*es. The *memtex* synchronizations are managed from the hypervisor, and triggered by a *hypercall* at each `__prem_notify()` call. The scheduler always allows the phase to execute at once, and as the hypercall dominates computation time, it is representative of the online cost for a pre-computed static schedule. We do not consider schedule-induced idle-time, as we are evaluating the effects of compiler transformation, and don't want scheduling policies to influence the results. Figure 5.11 shows that PREM only adds 16.3% to the execution time on average.

As in Section 4.2, the main contributor to the increased execution time is *preventive invalidation*, necessary to achieve the low amount of compute phase cache misses. The number of instructions executed also contributes to the final performance, as benchmarks with a low amount of instructions in the original program will be more sensitive to the cost of added instructions after transformation. No benchmark shows more than a 44% increase in execution time, which is acceptable to enabling predictable execution on COTS multi-core hardware. When considering the additional overhead of the scheduler, it is clear that enforcing memory protection at this level in the cache hierarchy leads to small scheduling overheads. The cost to perform the scheduling operation is very small compared to the amount of useful work that can be executed in intervals of $512KB/thread$. On average, the additional cost of the scheduling operations, hypercall included, is less than 3%. This again shows that the synchronizationi/scheduling

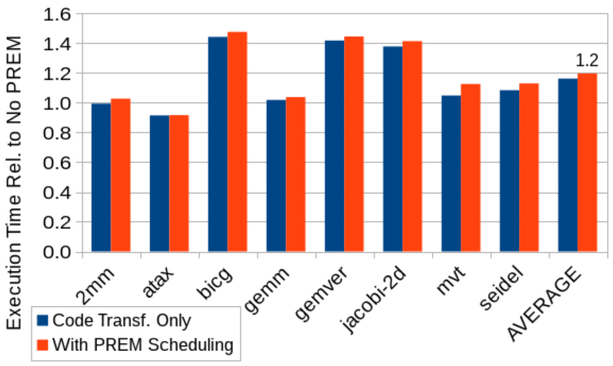


Figure 5.11: The relative performance of PREMized to non-PREMized tasks.

costs highlighted in the GPU evaluation in Section 4.1.2 are resolved by larger memories.

5.3 Conclusion

This section has addressed the question of cache behavior under PREM, and we have observed that the random replacement policies common in commercial off the shelf systems are a problem for predictable execution, as it introduces additional issues in ensuring that the memory prefetched during the PREM prefetch phases are indeed available locally during the compute phase.

In this chapter we have found that the impact of these effects can be managed by taking additional steps to ensure correct PREM operation. In the case of the GPU, we found that the use of the hardware managed caches required additional consideration on the sizing of PREM Intervals to ensure that the risk of data eviction before the compute phase was minimized. Fundamentally, we found that due to the configuration of the cache replacement policy, only three quarters of the cache sets were usable, and that the generated PREM intervals should be sized accordingly. In addition to this, we found that repetitions of the prefetch operations made sure that that the data was significantly less likely to be evicted before their use.

For caches on the CPU we did not see this particular characteristic, however, we found that the low size and associativity of caches play an important role in the ability of PREM intervals to reduce the CPMR. In particular, the L1 caches on the TX1 was subject to frequent self-evictions due to these factors, thus severely violating the PREM Isolation guarantees for several workloads. We were able to construct a well-performing PREM system by instead using the shared L2 cache, which had both larger size and associativity that prevented such self-evictions. However, to ensure that tasks executing on different cores were not able to interfere with each other, we demonstrated that this approach needed additional runtime support in the form of cache coloring to isolate the memory usage of the cores from each other.

All-in-all, the conclusion from this chapter is that larger local storage is better for PREM performance, reducing the impact of PREM synchronizations. In the evaluated platforms this could only be achieved by using hardware-managed caches, which we demonstrated to have significant issues with respect to PREM, both from a code generation and runtime isolation perspective. For this reason, we conclude that PREM *can* be used in cache-based systems if additional care is taken, but that the best fit for PREM is a system which has large software-managed SPM available. This is also in line with our findings on PREM execution on PULP in Section 4.3.

Chapter 6

System-Level Co-Scheduling Effects and Optimization for PREM

In the previous chapters we have explored the interaction between the PREM compiler, the PREM runtime, and PREM scheduling. In particular, in Section 4.2 we used an optimal ILP-based scheduler to produce an optimal schedule for the PREM system, based on intervals created by the compiler. However, PREM compilers and schedulers are individually unable to construct a well-optimized PREM system. The compiler has a local view of each task, which it can optimize for predictability and performance, but it cannot optimize for the interactions with other tasks deployed on the system. This is done by the scheduler, which has full visibility of all tasks in the system. However, even a state-of-the-art optimal scheduler may produce sub-optimal schedules, as the scheduler cannot improve the schedule beyond the degrees of freedom given by the intervals created by the compiler.

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

Soliman et al. [93] proposed to address this concern by integrating the scheduler into the compiler, but this solution is still subject to the local view of the compiler: All tasks need to be compiled together in a single compilation unit to enable scheduling, breaking common development flows such as partial compilation into object files. To fully address this issue, a new methodology that preserves the strengths of both the compiler's per-task and scheduler's per-system optimization, while finding the sweet spot is required.

In this chapter we will address this question, based on work done in collaboration with Maxim Mattheeuws, and published at *LCTES'20* [94]. In particular, we propose a novel methodology and prototype implementation for bridging the gap between the PREM compiler and the PREM scheduler, allowing us to reduce the response time of PREM systems by as much as 31%.

6.1 Problem Description

Our main goal is to co-select PREM intervals with the scheduling, such that we can minimize the Worst Case Response Time (WCRT) R of the system. We use the definition of the response time following from the PREM scheduling discussion in Section 1.4, and as shown in Equation 1.5. We see that there are three terms that should be minimized to reduce the response time R_τ of a PREM system; the blocking time on memory B^{memory}, the scheduling/synchronization cost $S(|I_\tau|)$ of the intervals I of task τ, and the execution time of the intervals e_τ. By reducing the response time, the performance of the system is improved, and it increases the likelihood of making the taskset T schedulable. The accumulated interval WCET e_τ is ideally constant[1], but the remaining terms can be tuned: To minimize B^{memory}, the interval lengths $len(i), i \in I_{\bar{\tau}}$ (where $\bar{\tau}$ means tasks other than τ) should be *minimized* to reduce blocking time. On the other hand, to minimize $S(|I_\tau|)$ the interval lengths $len(i), i \in I_\tau$ should be *maximized* to reduce the number of scheduling points and their overheads. We make two key observations:

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

Observation 2: The impact of both options on R_τ is only known after scheduling, and cannot be used by the compiler to select a different I' or to alter $size(i), i \in I$. Furthermore, the compiler can only infer information about the task τ under compilation, limiting optimizations to criteria available at the granularity at which compilers analyze programs (i.e., translation unit), effectively excluding optimizations to $\bar{\tau}$.

In light of these limitations, the previously presented PREM compiler has fallen back to select PREM intervals that are as large as possible, while still fitting into the local memory. This reduces the total amount of online scheduling decisions at runtime, and therefore $S(|I_\tau|)$, at the cost of increasing B^{memory} for other tasks. The compiler may therefore, by reducing the cost of $S(|I_\tau|)$ to optimize the performance for the task under compilation, increase the R^{memory} term resulting from scheduling, overall worsening R_τ.

Consider the example in Figure 6.1. In the top scenario, tasks τ_0 and τ_1 have been compiled with the larger-is-better interval sizing heuristic. While this is best for the performance of each individual task, the co-scheduled system suffers from serialization effects, as the memory phases (i.e., the prefetch and writeback phases) of τ_1 cannot be co-scheduled with those of τ_0. In this example, both tasks are released at time t_0 and the final phase of τ_1 finishes at t_4, which is the response time $R_{\{\tau_0,\tau_1\}}$ of the taskset $\{\tau_0, \tau_1\}$. In the bottom scenario,

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

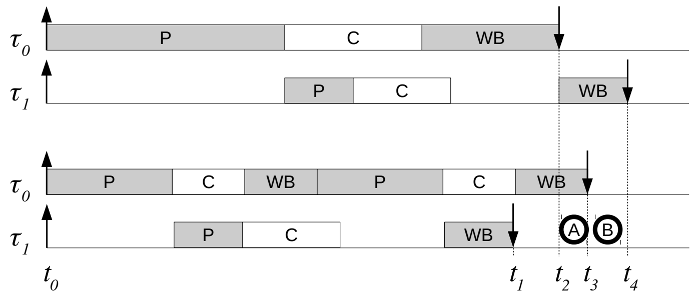


Figure 6.1: Illustrative example of how PREM interval sizing of individual tasks affects the overall system performance.

τ_0 has been divided into intervals of half the size. Due to scheduling overhead, the task now takes longer to execute, finishing at t_3 instead of t_2, marked Ⓐ. However, due to the finer scheduling granularity, τ_1 can now be scheduled earlier, completing already at t_1. Thus, while we increased the execution time of τ_0, the response time of the taskset $\{\tau_0, \tau_1\}$ is reduced from t_4 to t_3, marked Ⓑ.

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

6.2 Synergistic Methodology for PREM Compilation and Scheduling

For the first time, our methodology enables PREM tools, i.e., compiler, WCET analyzer, and scheduler to exchange information, as shown in Figure 6.2, to globally optimize the system. It can be fully automatized, driven by the novel *Optimizer* component. By using this methodology, the source code and real-time constraints are automatically transformed, analyzed, and scheduled to produce an optimized

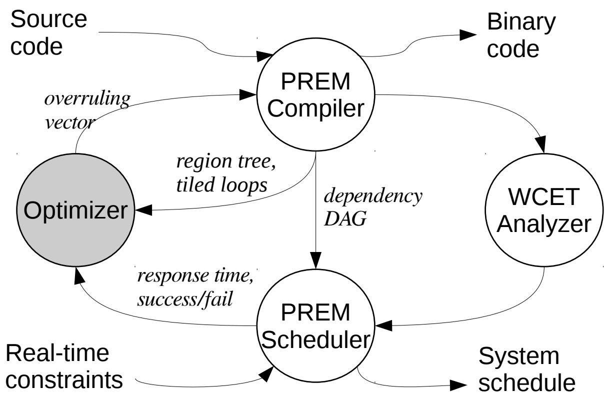


Figure 6.2: Data exchange in the proposed methodology.

PREM system, outputting executable binaries and a static system schedule.

The first steps of the methodology are directly derived from the traditional PREM system deployment approach: The first step of the proposed method is the compilation of the source code for each PREM task $\tau \in T$ with the PREM compiler to produce the PREM intervals I. As a starting point, the compiler relies on a simple heuristic that maximizes use of available local memory (to minimize synchronization overheads, $S(|I_\tau|)$). The compiled program is analyzed for WCET, i.e., upper bounds of the execution time of each PREM intervals $len(i)$ are derived. This can be done through measurements or using one of the static analysis methods available in the literature [9]. This provides the input to the PREM scheduler [31, 29, 36] together with the real-time constraints, e.g., the deadline D_τ. Additionally, as outlined in Section 3.2.4, the scheduler takes a directed acyclic graph (DAG) of the ordering of intervals as input, which is produced by the PREM compiler. The DAG specifies the dependencies between the PREM intervals to preserve program order in the constructed schedule. At the top level, the SESE edges e (corresponding to jumps j in Section 3.2.2) that connect the PREM intervals are transferred to the DAG, and within each intervals there are edges that specify the dependency of the writeback phase on the compute phase, and of the compute phase on the prefetch phase. The scheduler also gives additional information, e.g., a binary fail/success if the schedule respects the schedulability constraint as given in Equation 1.4, and the response time R of the system. This is where the traditional approach would end.

However, as outlined in Section 6.1, the identified solution may not be optimal with respect to R, as the schedule is only optimal for the exact set of intervals I produced by the compiler. If all real-time constraints are met (Equation 1.4), this might be acceptable, but if the scheduler was unable to schedule the selected intervals I without violating the constraints, this does not necessarily mean that the system is unschedulable under PREM.

6.2.1 Connecting the Region and Time Domains

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

Importantly, and as outlined in Section 1.4.1 and Chapter 3, the compiler operates in the *region domain* of the source code of the program, while the scheduler operates in the *time domain.* However, it is inadvisable to intertwine the fundamentally different *region* and *time domains* of the compiler and scheduler, as this increases tool complexity and interdependence, increasing the development and maintenance cost of either tool. In particular, the compiler should not be extended to be aware of the behavior of tasks outside its current compilation unit, and the scheduler should not be extended to understand the low-level code details of SESE regions in the *region domain.* The only shared concept that exist between the compiler and the scheduler are the PREM intervals I, which we instead use to pass information between the two.

To enable this, we introduce a new component, the *Optimizer* as shown in Figure 6.2 to manage the additional information. This optimizer is responsible for translating the output in the *time domain* from the scheduler, into a refined interval selection based on the regions Υ_τ (as defined in Section 3.2.2) of each task τ – thus ensuring that the compiler only handles the τ in the current translation unit.

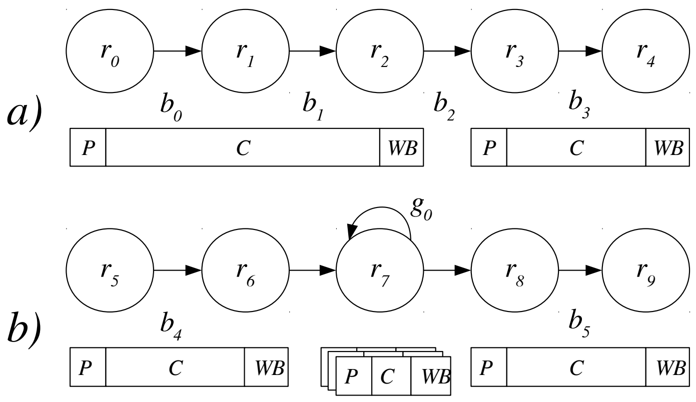


Figure 6.3: Overruling parameters linked to regions.

As shown in Figure 6.2, the PREM scheduler retains the same input and output parameters as defined in Sections 1.4.1, 3.2.4 and 6.2, but the output is in the proposed methodology redirected to the *optimizer*, which triggers a re-compilation and re-scheduling with a different set of intervals I' based on an *overruling vector* passed to the compiler. The *overruling vector* is given in terms of regions $r \in \Upsilon_\tau$ that the compiler natively understands, allowing it to alter the I_τ selected to improve the response time R achieved by the scheduler.

We will describe how the *optimizer* achieves this translation in Section 6.2.3, but first we define *overruling vectors*.

6.2.2 Overruling Vectors

The new *overruling vector* overrules the internal heuristics for interval selection in the compiler. We define the overruling vector Z as a collection of boolean values b, where each b represents a decision whether two regions $r_0, r_1 \in \Upsilon_\tau$ are to be combined into the same interval. Each b is mapped to the unique SESE edge e_{r_0,r_1} (see Section 3.2.2) connecting the SESE regions r_0 and r_1 in the CFG. If the boolean value b is *true* both regions r_0 and r_1 connected by this edge will be selected into the same interval i. Consider the example in Figure 6.3a: Five sequential regions r_0, r_1, r_2, r_3, r_4 are shown in the CFG. The figure depicts the interval selection that $\{b_0, b_1, b_2, b_3\} = \{1, 1, 0, 1\}$ would result in, giving two intervals $i_0 = \{r_0, r_1, r_2\}, i_1 = \{r_3, r_4\}$. This corresponds to the selection of regions r_{seq} in Section 3.2.2, which uses compiler heuristics to perform the same process. However, the overruling vector allows the proposed methodology to alter the interval selection of the built-in compiler heuristics to improve the overall system response time of all tasks, as we outlined in Section 6.1.

Regions with loops r_{loop} larger than local memory $size(r_{loop}) > size(\lambda)$ cannot be joined with other regions into an interval, as the loop region in itself is already too large, thus violating the constraint in Equation 1.1. This can intuitively be addressed by tiling and unrolling the loop region r_{loop} into smaller regions $r_{loopchunk_0}, r_{loopchunk_1}, \cdots$, and selecting each of them using the boolean values b for the resulting edges $e_{r_{loopchunk_0}, r_{loopchunk_1}}$ that connect them. This is the same effect as achieved by *loop tiling* (see Chapter 3), achieving the same effect but without unrolling the code and increasing the code size. In this case, the compiler will see a single region r_{loop}, and to inform the compiler that this region is to be split into tiles, we introduce the additional overruling vector L of tuples (r, g). Every tuple selects the *tiling granularity* g for the loop region r, determining how many iterations of the loop that are executed within each tile, as shown in Figure 6.3b. This constitutes a compact way of representing the unrolling operation that would otherwise be required, and corresponds to the interval heuristic presented in Sections 3.1.3 and 3.2.2.

Note that regions r_{loop} that are tiled can not be selected into an interval i with their predecessor and successor regions r_{pred} and r_{succ}. This, as the interval would without tiling violate the size constraint in Equation 1.1, and tiling invalidates the edges $e_{r_{pred}, r_{loop}}$ and $e_{r_{loop}, r_{succ}}$ (replacing them with multiple new edges). Therefore we do not associate a boolean value $b \in Z$ with edges connecting to a loop that must be tiled, allowing only the use of $(r, g) \in L$, as shown in Figure 6.3b. Here r_5 and r_6, as well as r_8 and r_9 retain the $b \in Z$ vector, while r_7 is only associated with $(r, g) \in L$. Note that the loop back-edge for region r_7 is placed outside the region for illustration purposes only. Note also that loop regions r_{loop} which are smaller than the local memory $size(r_{loop}) \leq size(\lambda)$ do not need to be tiled, but can be handled with the Z vector.

To guarantee that the resulting intervals from the overruling vectors L and Z still correspond to correct PREM intervals, we extend the compiler to validate that the overruled interval selection do not violate the constraint in Equation 1.1. This is achieved by performing a read-only run of the interval selection algorithm in Section 3.2.2, which implicitly does this for compiler-generated intervals.

6.2.3 Optimizer

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

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

Depending on the amount of metadata produced by the scheduler, the selection of which compiler decisions to *overrule* can be differently precise: If the scheduler outputs information about which intervals are blocking which, these can be selectively overruled, but in the generic case when no such information is provided, the *optimizer* must use a suitable algorithm to determine how to produce *overruling* vectors. We will discuss this further in Section 6.3.3, but first present a generic optimizer that makes as few assumptions as possible on the output of the scheduler.

Genetic PREM Optimizer: A Use Case

While the *optimizer* can be tailored to the unique characteristics of a specific PREM scheduler, we propose an *optimizer* that uses only the *response time* R of the system (a quantity that any PREM scheduler will output) to implement the proposed methodology. As small changes in the interval sizing could potentially significantly alter the optimal interleaving of memory phases, we are searching for a global optimum on a non-continuous optimization function. For this reason, we base our optimizer on a genetic algorithm (GA) [95], as they are known to perform well on such functions [96]. GAs operate on *populations* of N *individuals* and execute in epochs. At the end of each epoch the *fitness* F of the population is evaluated, in our algorithm given by $F = \frac{1}{R}$, where R is the response time. Each individual in the population has a different *genome* γ that represents the characteristic of the individual. In our algorithm, the genome is represented by the *overruling vectors* Z and L for every task in the taskset, as follows: Each $b \in Z$ is a single bit representing the decision whether two regions are to be combined into the same interval. Each $g \in L$ represents the tiling granularity and uses as many bits as required to express the maximum tiling granularity that results in an interval smaller than the local memory. This value is implicitly provided by the compiler during the first epoch, as the intervals are then generated by the compiler heuristic of selecting the largest possible intervals – and as already stated, loop intervals that require tiling will always be represented as a separate interval.

As such, we can represent the parameters L and Z, which describe each possible interval selection of the task (e.g., all *individuals*), as a single binary string, which maps well to GAs. Reusing the example from Figure 6.3, example (a) would use four bits to represent b_0, b_1, b_2, b_3, and example (b) would use two bits to represent b_4, b_5, as well as the bits required to represent g_0. If the maximum legal value of g_0 is 432, an additional 9 bits would be used to represent $base_2(g_0) = 110110000$. Any g not on the form $2^n - 1$ can express a tiling granularity that would result in intervals larger than $size(\lambda)$, violating Equation 1.1, and we assign these a fitness score of 0 when caught by the check in the compiler.

Table 6.1: Benchmarks of different memory complexity.

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

At the end of each epoch *individuals* with the worst fitness F are eliminated, determined by the generation gap parameter G. We use $G = 0.5$, meaning 50% of the individuals are eliminated. Following this, new *individuals* are generated through two processes: *Crossover*, in which the genome of two surviving individuals are combined into a new individual, and *mutation*, where a new individual is generated by randomly changing the genome of an individual [95]. The former explores solutions close to known good solutions as well as their linear combinations, and how many individuals are affected is determined by the crossover rate C. The latter introduces random variance into the genome by randomly flipping a bit in the genome γ so that the algorithm is not stuck in a local optimum. How often this occurs is determined by the mutation rate M.

6.3 Evaluation

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

To investigate memory blocking, we use benchmarks of different compute-to-communication ratios (CCR), which block the memory for different amounts of time. We initially consider Basic Linear Algebra Subroutines (BLAS) [97], which are classified according to their CCR, as shown in the first three rows of Table 6.1: The memory-bound BLAS1 kernel *axpy*, the BLAS2 kernel *gemv*, and the compute-bound BLAS3 kernel *gemm*. We refer to these kernels as AY, GV, and GM.

We use the optimizer from Section 6.2.3 and implement *overruling vectors* in previously presented compilers [32, 36]. Evaluation of PREM schedulers and WCET analyzers is out of scope of this work, and we use a first-come-first-served (FCFS) schedule and measurement-based execution times. As the benchmarks are loop-based, they always execute in a steady-state of repeating fixed sized intervals under FCFS.

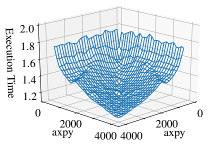
We implement PREM runtime scheduling in a separate process, which we call the *memory arbitrator* (MA), on one of the unused cores. The MA ensures that only a single task executes a memory phase at a time. At the start of each experiment, the PREM tasks use a UNIX socket to connect to the MA to set up a shared memory region *shm* (not included in measurements). Shared memory is the fastest inter-process communication in the system[2], but still subject to overheads, as will be shown in Section 6.3.2. Each PREM phase starts with a handshake with the MA using *shm*, and only when the arbitrator gives permission the task executes the phase.

6.3.1 Evaluation Platform and Setup

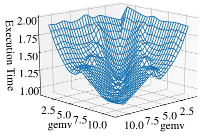
We evaluate on the NVIDIA TX2 [98] SoC, featuring a 4-core ARM A57 cluster, each core with a 32 KB private cache, and a 2 MB shared L2. All cores share the global memory. For our PREM setup, we use the A57 cores, dimensioning the PREM intervals to stage data through the private L1. The TX2 runs Ubuntu Linux 16.04, and to avoid OS scheduling interference, we migrate all processes that are not under investigation to a single core. As the tasks are mostly sleeping, their impact on the memory system is negligible.

We execute a PREM task on each of the two remaining cores. For each task τ executed in the experiments (i.e., an instance of AY, GV, or GM), we select input sizes such that each task has the same execution time R_{base}. This provides an intuitive measure of how well the memory blocking time B^{memory} is minimized: If we co-run two

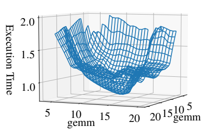
[2]Found using `ipc-bench` (github.com/goldsborough/ipc-bench).



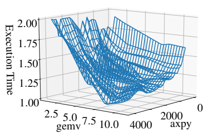
(a) AY×AY.



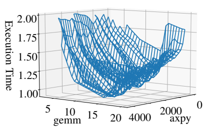
(b) GV×GV.



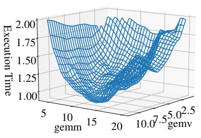
(c) GM×GM.



(d) AY×GV.



(e) AY×GM.



(f) GV×GM.

Figure 6.4: Execution times of the *symmetric* and *asymmetric* scenarios.

tasks that in isolation require R_{base} time units to finish, the time to run the two tasks τ_0 and τ_1 in parallel R_{corun} (i.e., the response time of the taskset) would be the same $R_{corun} = R_{base}$ time units if the tasks never block each other. We can then quantify the optimality score OS of the interval selection as $OS = \frac{R_{base}}{R_{corun}}$. If the tasks never block each other, we would get $OS = 1.0$, while complete serialization gives $OS = 0.5$. For our experiments we select input sizes such that for each task $R^{\tau}_{base} \approx 0.8s$, and for each taskset $R^{\tau_0,\tau_1}_{base} = max(R^{\tau_0}_{base}, R^{\tau_1}_{base})$.

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

6.3.2 Solution space exploration

For each scenario we plot the execution times for the exhaustive exploration of the selection space in Figure 6.4, using the *tiling granularity* g of the BLAS kernel loop to represent the selection space of each task. As the benchmarks are loop-bound, the effect of the interval *overruling* parameter L will be most dominant, and we will discuss the impact of the Z vector in Section 6.3.2. As outlined in Section 6.1, the compiler heuristics produce the largest intervals possible, represented by the largest g in each dimension.

The results of the symmetric scenarios AY×AY, GV×GV, and GM×GM are shown in Figures 6.4a, 6.4b, and 6.4c. They are symmetrical around the $X = Y$ plane, which is expected as $\{\tau_0, \tau_1\}$ achieves the same score as $\{\tau_1, \tau_0\}$ since both tasks are the same. With decreasing loop granularities g, the taskset response time R_{corun} generally increases, as the fixed-size scheduling cost (i.e., $S(|I_\tau|)$, see Equation 1.5) becomes more and more dominant. Note that to increase readability, we are not plotting configurations whose response time $R_{corun} > 2s$. However, in AY×AY and GV×GV local minima are exposed when the tiling granularity g_i of τ_i is a multiple of g_j of τ_j. As AY has a lower CCR, its intervals are shorter and more sensitive to this overhead, causing an offset in the minimum. In GM×GM, this effect is only a plateau, due to the higher CCR causing less memory contention.

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

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

Table 6.2: Difference between the best and worst configurations of the Z vector, in percent of the response time R.

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

The results for the asymmetric scenarios AY×GV, AY×GM, and GV×GM are shown in Figures 6.4d, 6.4e, and 6.4f. These plots have a much more angular surface due to local optima where an even number of intervals of τ_0 can be executed during an interval of τ_1, or vice versa. In scenarios with GM, the best OS is always achieved when $g_{GM} = 15$ (as found before). For the co-running task however, we see two different effects for AY and GV. The GV×GM scenario is compute-bound enough to not introduce any significant memory blocking, and for GV the larger-is-better compiler heuristic leads to the best optimality score $OS = \frac{0.83}{0.90} = 0.92$. In the AY×GM scenario, however, R_{corun} can be reduced compared to the compiler selected $g_{AY} = 4096$ to $g_{AY} = 3369$. This implies a reduction by the tile size of $\sim 1/5$th, increasing the OS of AY×GM from $OS = \frac{0.80}{1.35} = 0.59$ to $OS = \frac{0.80}{0.93} = 0.86$, providing a 45% increase in the optimality score.

For AY×GV, reducing the tiling granularity g of both tasks yields the best result. Reducing the compiler selected $g_{AY} = 4096; g_{GV} = 11$ to $g_{AY} = 3830; g_{GV} = 8$ increases the optimality score from $OS = \frac{0.80}{1.12} = 0.71$ to $OS = \frac{0.80}{1.0} = 0.80$. This shows that reduced interval sizes can reduce the total response time R_{corun}, at the cost of task performance (due to increased $S(|I_\tau|)$), as illustrated in AY×GV: For the best version under co-scheduling, the per-task execution times were increased by 1% for AY and 4.2% for GV, due to finer-grained scheduling. However, this reduces the response time R_{corun} of the co-scheduled taskset by 11%.

Impact of Z overruling vectors

As the AY, GV, GM benchmarks are primarily loop based, the impact of the loop overruling vectors L are the most dominant. The impact of the combination overruling vector Z on the total execution time is presented in Table 6.2. While the impact is relatively small (the Z vector can only affect the execution time by a few percent), it might

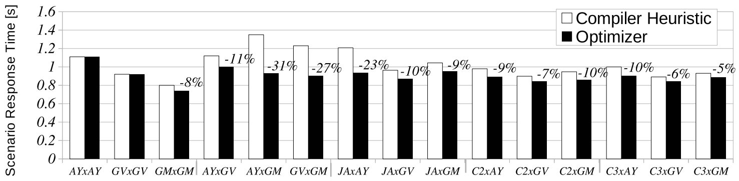


Figure 6.5: The performance of previously proposed compiler heuristics compared to the best optimized solution.

not be negligible in optimizing a system with tight deadlines D_τ.

Generally for three-phase PREM intervals, the behavior of loop-based and more sequential programs will be similar, as the L optimization vector can be expanded into the Z vector through unrolling, as outlined in Section 6.2.2. The impact these intervals have on the memory blocking time B^{memory} is limited by the size of the local memory $size(\lambda)$, as all intervals, unrolled or not, must conform to the requirement of Equation 1.1. Therefore, these results are also representative for the impact of the Z vector in non-loop based programs.

However, the PREM model also supports single-phased *compatible intervals* [24] to execute portions of the code which can not be transformed by the compiler (e.g., *syscalls*) as a single memory phase. These intervals take ownership of the memory system during their entire execution (retaining the PREM *single-core equivalence* property), affecting B^{memory}. As both memory and computation time contribute to their $len(i)$, it is not limited by $size(\lambda)$. The Z vector will start to dominate the L vector as the amount of *compatible intervals* with long execution times $len(i)$ increases. As *compatible intervals* should be avoided due to their bad behavior on B^{memory}, they are not otherwise covered in this paper.

Performance of Optimizer

To reach the described optima, the GA used the following parameters: Crossover rate $C = 12.5\%$, mutation rate $M = 12.5\%$, and population size of $N = 100$. The GA used a scaling window $W = 1$ and a pure selection strategy P. These parameters are chosen based on the discussion by Grefenstette [95]. We observe rapid convergence: on average 91% of the improvement was achieved already in the first epoch of the GA (worst being 88%). In this setup, 40 epochs take 6.5 hours on the TX2 on average. Within 10 epochs we achieved results within 3% of the best with only $N = 32$, which complete in under an hour. In larger PREM systems, scheduling more than two tasks will require more time to complete, and an optimization of the GA parameters and the workstation could be appropriate. However, we argue that the time to solution is tractable, as the optimizer is only invoked once. Furthermore, each task within the system can be developed and tested with short iteration times, and only when all tasks are final, the proposed methodology is applied.

The presented GA-based *optimizer* uses only the response time R as optimization criteria. As shown, this is sufficient to solve the problem outlined in Section 6.1, and as it is a fundamental output parameter of any scheduler, it has the additional benefit of working with any PREM scheduler proposed in the literature. However, with tighter coupling of the *optimizer* to the scheduler, a more specialized *optimizer* could be possible. As part of our ongoing work we are exploring the explicit targeting of the intervals $i \in I_\tau$ that cause the maximum memory blocking time B^{memory} for other tasks $\bar{\tau}$ in the system. By focusing the optimization on this task τ it might be possible to reduce the time to solution.

6.3.3 Compiler vs Optimizer

Having determined that 10 epochs with $N = 32$ gives results close to the optima, we use these parameters to extend the evaluation to include three stencil kernels, *jacobi-1d* (JA), *convolution-2d* (C2) and *convolution-3d* (C3) (as shown in the bottom three rows of Table 6.1) from the PolyBench suite [73]. These benchmarks are executed in scenarios with the previously presented BLAS kernels, the results of which are shown in Figure 6.5. The first six set of bars refer to the BLAS scenarios described earlier.

The results show that the proposed optimizer can reduce the scenario response times up to 31% over the compiler generated intervals. Two lessons can be learned from this. First, once intervals are large enough to dominate the scheduling cost $S(|I_\tau|)$, it is more efficient to optimize for locality rather than maximizing interval size. This was shown clearly in the performance gain in the GM kernel in Figure 6.4c. As it only affects the task τ currently being compiled, is an optimization that could be implemented with compiler heuristics. Second, we validate that selecting the maximum PREM interval sizes can cause significant memory blocking B^{memory}, negatively impacting the response time R_τ as shown in in Equation 1.5. By selecting smaller interval sizes, the interleaving of memory and compute phases could be improved between the two tasks, as suggested by the motivating example in Figure 6.1, leading to a reduction in the response time R_{corun} in all but two scenarios.

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

6.4 Conclusion

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

Chapter 7

Conclusion

In this thesis we have presented the first thorough exploration of the Predictable Execution Model on real systems. From the original presentation of PREM by Pellizzoni et al [24] – which has received over 200 citations since its publication in 2011 – and forward a vast majority [28, 29, 30, 31, 27, 26] on the work on PREM have been focused on the scheduling aspect, but very few publications have taken the step to run PREM-compatible tasks on real platforms. The publications that make up this thesis [39, 57, 38, 40, 94, 82, 36, 79, 48, 69] account for a large share of such publications in the scientific literature.

A key enabler in large scale experimentation of any software system is the availability of adequate compiler support. The question of PREM compilation has been thoroughly addressed in Chapter 3, which has explored which analyses that are required to extract the necessary information from legacy source code, which limitations that this imposes on the source code, and how to transform the code to perform well on a diverse set of platforms. Several interesting engineering questions remain to make the PREM compiler all that it can be, e.g., by introducing more sophisticated and state of the art algorithms for each individual aspect of the flow. Clear candidates for such exercises are better tiling algorithms and SPM allocation mechanisms. The pursuit of such engieering effort in these subproblem areas has been out of scope of this thesis, as the steps to do so are well documented in other areas of research. In fact, each such subproblem in compiler design has a full fledged field of research around it. Instead, we have shown that the proposed blocks of the compiler are necessary and sufficient to produce PREM code from real code for real systems.

While the compiler presented in this thesis has been used for most explorations of PREM on real hardware in the scientific literature, it is not the only PREM compiler project. In parallel to our efforts another PREM compiler has been deveoped at the University of Waterloo. This work is to a large extent complementary to the work presented in this thesis, as it has explored alternative implementations for SPM allocation [32], interval selection [93], and scheduling [99]. This increased breadth in the compiler-internal techniques provides a fertile ground for the next stage of PREM systems research. Collaborations on such projects are under planning, and promise further advancement in this field of research.

From the important work that now lies behind us, we have been able to establish that the three-interval Predictable Execution Model approach does indeed fulfill its promises outside of a controlled laboratory simulator setting. In Chapter 2 we performed an initial exploration on the applicability of PREM to heterogeneous systems, one of the first of its kind – other PREM explorations at the time were done by Paolo Burgio et al [28] and Capodieci et al [20] – which resulted in the creation of GPUguard. GPUguard showed how PREM scheduling could be introduced within the limited execution model of GPUs, without access to event and timer triggered executions, and extending the control of the PREM scheduler from the CPU OS to the full heterogeneous system. The lessons drawn from creating PREM-enabled GPU kernels, in the form of a UAV path planner, provided the necessary input for the first steps of the compiler implementation.

Following the exploration of PREM compiler techniques in Chapter 3, Chapter 4 explored the impact of the PREM transformations on co-executing tasks on three different platforms. We showed that the compiler techniques presented fulfilled the goal of *robustness to interference*, and that the produced PREM-enabled versions of the code in many cases performed better or on-par with their non-PREM counterparts. For GPU execution we showed that we could achieve PREM execution in a heterogeneous setting with as little as 1.2% overhead compared to the non-PREM implementation – a very small price to pay for *robustness to interference.* On the CPU, we showed that when the PREM code generation presented in Chapter 3 was coupled with PREM schedulers from CTU Prague, we could consistently produce static PREM schedules of the compiler-produced PREM intervals that had tighter worst-case response times – up to 45% – than that of legacy approaches, and that the PREM versions never exceeded the WCET guarantees computed offline. Lastly, we performed an initial exploration of PREM on PULP, where we determined that the similarity of the three-phase prefetch-compute-writeback PREM intervals to the native SPM+DMA execution model was able to automatically generate code that performed well on the platform, where traditionally significant manual labor would have been required. In doing this, we showed that *robustness to freedom* could be achieved at or at near-expert programmer performance, with improved portability of code as a positive side-effect.

In the cases where the performance of the PREM versions of the programs would not be competitive to their legacy counterparts, we explored and documented the causes. In particular, we determined that the size of the local memory in connection with the cost of cross-device synchronization and the frequency at which such scheduling can take place is a key determiner in the performance of PREM. Frequent refills of the local memory lead to significant contention not for the memory system but for mutual exclusive access to the memory system. We further documented how the unpredictable replacement policies of caches can negatively impact PREM performance, due to additional steps required to ensure that no cache misses occur during the compute phase.

In Chapter 5 we explored these limitations in greater detail, and showed how they could be lifted or worked around. In particular, we showed how the overheads due to small local memories and their frequent refills on GPUs could be worked around by using larger hardware-managed caches in a predictable way. We determined what aspects of the cache replacement policy – the non-uniform set eviction probabilities – that were causing issues with the PREM isolation, and presented a technique to address this. At the end we could demonstrate correct PREM execution on hardware caches for kernels which were previously shown to perform badly on the SPMs – issues that could be addressed without changing hardware platform using the presented techniques. Furthermore, we explored the impact of cache sizes and associativity with PREM execution on CPUs. On our experimental system the core private L1 caches, that would have been ideal from an isolation perspective, were shown to have significant drawbacks with frequent self-evictions due to low associativity and size. Again, without having to change platform, we could ensure that tasks were executing in accordance with the *robustness to interference* guarantees by changing to the larger last-level cache, and applying orthogonal cache locking (in the form of cache coloring, due to missing hardware support for locking cache lines) techniques to achieve the required isolation guarantess.

Finally, in Chapter 6 we showed that the efficient code generation presented in the previous chapters would not be enough by themselves to produce optimal PREM systems – due to the dependency on the runtime scheduling effects of mutual exclusion of memory phases. Presented PREM scheduling techniques operate under the assumption that the PREM intervals are fixed in size, and cannot be changed, thereby producing suboptimal systems due to unneeded memory blocking. We proposed a methodology to address this issue, which makes PREM compiler and PREM scheduler able to exchange information across their very different domains, and achieve schedules that were up to 31% better than what the scheduler alone would have been able to produce.

Summarizing, this thesis has provided the first broad scale exploration of the behavior of PREM on real platforms, enabled by efficient compiler support. This support has allowed us to identify several weaknesses and propose solutions.

Several future directions lie open for exploration, including closer integration with processes within the industry, where techniques such as PREM would supposedly be applied. A clear example would be the integration with Domain Specific Modeling Languages (DSMLs) such as AUTOSAR, which provide their own abstractions for task scheduling and exchange of data. By integrating the PREM compiler with a PREM-aware code generator for AUTOSAR systems, PREM could be made even more attractive in practice.

In the same sprit, the implementation of improved internal compiler components – tilers, SPM allocators – would increase the benefits of automatic code generation. An area of high interest for future engineering endeavors would also be to start lifting some of the restrictions on compiled code that were outlined in Section 3.3, in particular related to relaxing constraints on non-symbolic expressions for loop bounds as a prerequisite for tiling and the creation of predictable intervals. This is a requirement that falls just outside otherwise standard real-time coding guidelines, and would technically be possible to lift, although it was out of scope for this work.

To conclude, a final direction that would be interesting to explore in further detail would be to apply the techniques developed as part of this thesis outside the scope of hard-real time applications. As shown in the evaluation of PREM on PULP, the techniques developed to support PREM are closely related to performance optimizations for memory-bottlenecked systems. As the speed of computation has gone up, most systems today are bottlenecked by the memory systems, as data simply cannot be delivered to the processor fast enough. By applying the techniques outlined in this thesis, the data locality optimizations could be repurposed to increase performance in everything from soft-realtime embedded systems to distributed high-performance computing platforms.

The future lies wide open.

Bibliography

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

[2] R. Cavicchioli, N. Capodieci, and M. Bertogna, “Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms,” in *ETFA’17*, 2017.

[3] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen, “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. [Online]. Available: doi.ieeecomputersociety.org/10.1109/TPDS.2016.2586074

[4] A. Skende, “Introducing parker: Next-generation tegra system-on-chip,” in *2016 IEEE Hot Chips 28 Symposium (HCS)*, Aug 2016.

[5] “Amd embedded g-series lx,” Mar 2018. [Online]. Available: https://www.amd.com/en/products/embedded-g-series-lx

[6] S. Mittal and J. S. Vetter, “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. [Online]. Available: http://doi.acm.org/10.1145/2636342

[7] M. Dehyadegari, A. Marongiu, M. R. Kakoee, S. Mohammadi, N. Yazdani, and L. Benini, “Architecture support for tightly-coupled multi-core clusters with shared-memory HW accelera-

tors," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2132–2144, Aug 2015.

[8] J. Liu, *Real-Time Systems.* Prentice Hall, 2000. [Online]. Available: https://books.google.ch/books?id=855QAAAAMAAJ

[9] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem: Overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1347375.1347389

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

[11] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens, "A globally arbitrated memory tree for mixed-time-criticality systems," *IEEE Trans. on Computers*, vol. 66, no. 2, Feb 2017.

[12] D. Dasari, V. Nelis, and B. Akesson, "A framework for memory contention analysis in multi-core platforms," *Real-Time Systems*, vol. 52, no. 3, May 2016.

[13] I. Agirre, J. Abella, M. Azkarate-Askasua, and F. J. Cazorla, "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.

[14] S. Girbal, X. Jean, J. Le Rhun, D. G. Pérez, and M. Gatti, "Deterministic platform software for hard real-time systems using multi-core cots," in *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th.* IEEE, 2015, pp. 8D4–1.

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

[16] H. Yun, W. Ali, S. Gondi, and S. Biswas, "BWLOCK: A dynamic memory access control framework for soft real-time applications on multicore platforms," *IEEE Trans. on Computers*, vol. 66, no. 7, 2017.

[17] G. A. Elliott, B. C. Ward, and J. H. Anderson, "Gpusync: A framework for real-time gpu management," in *2013 IEEE 34th Real-Time Systems Symp.*, 2013.

[18] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," in *ASPLOS'16*. ACM, 2016.

[19] W. Ali and H. Yun, "Protecting Real-Time GPU Applications on Integrated CPU-GPU SoC Platforms," *ArXiv e-prints*, Dec. 2017.

[20] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna, "Sigamma: Server based integrated gpu arbitration mechanism for memory accesses," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS '17. New York, NY, USA: ACM, 2017, pp. 48–57. [Online]. Available: http://doi.acm.org/10.1145/3139258.3139270

[21] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in COTS-based multi-core systems," in *RTAS'14*, 2014.

[22] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee, "Response time analysis of COTS-based multicores considering the contention on the shared memory bus," in *Trust-Com'19*, 2011.

[23] S. Saidi and A. Syring, "Exploiting locality for the performance analysis of shared memory systems in MPSoCs," in *RTSS'18*, 2018.

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

[25] C. M. Kirsch and A. Sokolova, “The logical execution time paradigm,” in *Advances in Real-Time Systems.* Springer, 2012, pp. 103–120.

[26] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo, “Memory-aware scheduling of multicore task sets for real-time systems,” in *RTCSA’12*, 2012.

[27] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, “Memory-centric scheduling for multicore hard real-time systems,” *Real-Time Systems*, vol. 48, no. 6, pp. 681–715, 2012.

[28] A. Alhammad and R. Pellizzoni, “Time-predictable execution of multithreaded applications on multicore systems,” in *DATE’14*, 2014.

[29] ——, “Schedulability analysis of global memory-predictable scheduling,” in *EMSOFT’14*, 2014.

[30] A. Alhammad, S. Wasly, and R. Pellizzoni, “Memory efficient global scheduling of real-time tasks,” in *RTAS’15*, 2015.

[31] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo, “Global real-time memory-centric scheduling for multicore systems,” *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2739–2751, Sep. 2016.

[32] M. R. Soliman and R. Pellizzoni, “WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching,” in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Bertogna, Ed., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 24:1–24:23. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7175

[33] R. Mancuso, R. Dudko, and M. Caccamo, “Light-prem: Automated software refactoring for predictable execution on cots embedded systems,” in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications.* IEEE, 2014, pp. 1–10.

[34] P. Burgio, A. Marongiu, P. Valente, and M. Bertogna, "A memory-centric approach to enable timing-predictability within embedded many-core accelerators," in *RTEST'15*. IEEE, 2015.

[35] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. yi, "A survey on static cache analysis for real-time systems," 2016.

[36] J. Matějka, B. Forsberg, M. Sojka, P. Sucha, L. Benini, A. Marongiu, and Z. Hanzálek, "Combining PREM compilation and static scheduling for high-performance and predictable MP-SoC execution," *Parallel Computing*, 2019.

[37] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," in *RTAS'19*, 2019.

[38] B. Forsberg, L. Benini, and A. Marongiu, "Taming data caches for predictable execution on gpu-based socs," in *DATE'19*, 2019.

[39] B. Forsberg, A. Marongiu, and L. Benini, "Gpuguard: Towards supporting a predictable execution model for heterogeneous soc," in *DATE'17*, 2017.

[40] B. Forsberg, L. Benini, and A. Marongiu, "Heprem: A predictable execution model for gpu-based heterogeneous socs," *IEEE Transactions on Computers*, 2020.

[41] "NVIDIA Jetson TX1 Module." [Online]. Available: https://developer.nvidia.com/embedded/jetson-tx1

[42] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 105–118, 2011.

[43] NVIDIA Corporation, "NVIDIA CUDA C++ programming guide," 2020, version 11.1. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[44] ——, "JetPack 3.2.1 release notes." [Online]. Available: https://developer.nvidia.com/embedded/jetpack-3_2_1

[45] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, *Introduction to autonomous mobile robots.* MIT press, 2011.

[46] D. Watzenig and M. Horn, *Introduction to Automated Driving.* Springer International, 2017.

[47] P. Burgio, M. Bertogna, I. S. Olmedo, P. Gai, A. Marongiu, and M. Sojka, "A software stack for next-generation automotive systems on many-core heterogeneous platforms," in *Digital System Design (DSD).* IEEE, 2016.

[48] B. Forsberg, D. Palossi, A. Marongiu, and L. Benini, "Gpu-accelerated real-time path planning and the predictable execution model," *Procedia Computer Science*, vol. 108, pp. 2428 – 2432, 2017, international Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050917308256

[49] D. Palossi, M. Furci, R. Naldi, A. Marongiu, L. Marconi, and L. Benini, "An energy-efficient parallel algorithm for real-time near-optimal uav path planning," in *Proceedings of the ACM International Conference on Computing Frontiers.* ACM, 2016.

[50] C. G. Cassandras and S. Lafortune, *Introduction to discrete event systems.* Springer Science & Business Media, 2009.

[51] E. Dijkstra, "A note on two problems in connexion with graph," *Numerische Mathematik*, 1959.

[52] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, 2001.

[53] M. Bauer, H. Cook, and B. Khailany, "Cudadma: optimizing gpu memory bandwidth via warp specialization," in *High performance computing, networking, storage and analysis.* ACM, 2011.

[54] D. Palossi, A. Marongiu, and L. Benini, "On the accuracy of near-optimal gpu-based path planning for uavs," in *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 85–88. [Online]. Available: https://doi.org/10.1145/3078659.3079072

[55] stress(1) - linux man page. [Online]. Available: https://linux.die.net/man/1/stress

[56] R. Mancuso, R. Dudko, and M. Caccamo, "Light-prem: Automated software refactoring for predictable execution on cots embedded systems," in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2014, pp. 1–10.

[57] B. Forsberg, L. Benini, and A. Marongiu, "HePREM: Enabling predictable GPU execution on heterogeneous SoC," in *DATE'18*, 2018.

[58] 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

[59] OpenACC-Standard.org, "The openacc application programming interface version 2.6," Nov 2017. [Online]. Available: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf

[60] J. Bosch, A. Filgueras, M. Vidal, D. Jimenez-Gonzalez, C. Alvarez, and X. Martorell, "Exploiting parallelism on gpus and fpgas with ompss," in *Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC Systems*, ser. ANDARE '17. New York, NY, USA: ACM, 2017, pp. 4:1–4:5. [Online]. Available: http://doi.acm.org/10.1145/3152821.3152880

[61] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[62] *MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems.* Motor Industry Research Association, 2013.

[63] "GCC, the GNU compiler collection," 2020. [Online]. Available: https://gcc.gnu.org

[64] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, March 2004, pp. 75–86.

[65] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, T. Chen, H. Sung, C. Bertolli, and K. O'Brien, "Offloading support for openmp in clang and llvm," in *LLVM-HPC'16*, 2016.

[66] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly — performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, 2012.

[67] S. Sioutas, S. Stuijk, H. Corporaal, T. Basten, and L. Somers, "Loop transformations leveraging hardware prefetching," February 2018.

[68] K. Koukos, P. Ekemark, G. Zacharopoulos, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Multiversioned decoupled access-execute: The key to energy-efficient compilation of general-purpose programs," in *CC'2016.* ACM, 2016, pp. 121–131.

[69] B. Forsberg, M. Solieri, M. Bertogna, L. Benini, and A. Marongiu, "The predictable execution model in practice: Compiling real applications for cots hardware," submitted to TECS.

[70] M. A. Serrano, S. Royuela, and E. Quiñones, "Towards an openmp specification for critical real-time systems," in *Evolving OpenMP for Evolving Architectures*, B. R. de Supinski, P. Valero-Lara, X. Martorell, S. Mateo Bellido, and J. Labarta, Eds. Cham: Springer International Publishing, 2018, pp. 143–159.

[71] R. Tabish, R. Mancuso, S. Wasly, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A reliable and predictable scratchpad-centric os for multi-core embedded systems," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 377–388.

[72] J. Absar, "Scalar evolution - demystified," in *European LLVM Developers Meeting*, 2018.

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

[74] B. Cyrill, "Dma management for predictable execution on parallel many-core accelerators," Master's thesis, ETH Zürich, Department of Information Technology and Electrical Engineering, Zurich, Switzerland, 2019.

[75] K.-A. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Clairvoyance: Look-ahead compile-time scheduling," in *CGO'17*. IEEE, 2017, pp. 171–184.

[76] A. Kurth, P. Vogel, A. Marongiu, and L. Benini, "Scalable and efficient virtual memory sharing in heterogeneous socs with tlb prefetching and mmu-aware dma engine," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 292–300.

[77] G.-T. Bercea, C. Bertolli, S. F. Antao, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, and K. O'Brien, "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*, ser. PMBS '15. New York, NY, USA: ACM, 2015, pp. 2:1–2:11. [Online]. Available: http://doi.acm.org/10.1145/2832087.2832089

[78] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, 2012.

[79] B. Forsberg, L. Benini, and A. Marongiu, “On the cost of freedom from interference in heterogeneous socs,” in *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 31–34. [Online]. Available: https://doi.org/10.1145/3207719.3207735

[80] NVIDIA, “Cuda c programming guide v9.1.85,” March 2018. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[81] X. Mei and X. Chu, “Dissecting gpu memory hierarchy through microbenchmarking,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, Jan 2017.

[82] J. Matějka, B. Forsberg, M. Sojka, Z. Hanzálek, L. Benini, and A. Marongiu, “Combining prem compilation and ilp scheduling for high-performance and predictable mpsoc execution,” in *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 11–20. [Online]. Available: https://doi.org/10.1145/3178442.3178444

[83] Z. Hanzálek and P. Šůcha, “Time symmetry of resource constrained project scheduling with general temporal constraints and take-give resources,” *Annals of Operations Research*, vol. 248, no. 1, pp. 209–237, Jan 2017. [Online]. Available: https://doi.org/10.1007/s10479-016-2184-6

[84] J. F. Henriques, R. Caseiro, P. Martins, and J. Batista, “High-speed tracking with kernelized correlation filters,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 3, pp. 583–596, 2014.

[85] A. Kurth, P. Vogel, A. Capotondi, A. Marongiu, and L. Benini, “HERO: Heterogeneous embedded research platform for exploring RISC-V manycore accelerators on FPGA,” *arXiv*, 2017.

[86] A. Pullini, D. Rossi, I. Loi, G. Tagliavini, and L. Benini, “Mr.wolf: An energy-precision scalable parallel ultra low power soc for iot

edge processing," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 7, pp. 1970–1981, 2019.

[87] RI5CY: RISC-V core. [Online]. Available: https://github.com/pulp-platform/riscv

[88] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc.* Strathclyde Academic Media, 2014.

[89] J. D. Bakos, "Chapter 1 - the Linux/ARM embedded platform," in *Embedded Systems*, J. D. Bakos, Ed. Boston: Morgan Kaufmann, 2016.

[90] M. Mattheeuws, Master's thesis, ETH Zürich, Department of Information Technology and Electrical Engineering, Zurich, Switzerland, 2019.

[91] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no VM exits! (almost)," *CoRR*, 2017.

[92] I. Sañudo, P. Cortimiglia, L. Miccio, M. Solieri, P. Burgio, C. Di Biagio, F. Felici, G. Nuzzo, and M. Bertogna, "The key role of memory in next-generation embedded systems for military applications," in *SEDA'2018'.* Springer Int., 2018.

[93] M. R. Soliman and R. Pellizzoni, "Prem-based optimal task segmentation under fixed priority scheduling," in *2019 31st Euromicro Conference on Real-Time Systems (ECRTS)*, 2019, pp. 1–24.

[94] B. Forsberg, M. Mattheeuws, A. Kurth, A. Marongiu, and L. Benini, "A synergistic approach to predictable compilation and scheduling on commodity multi-cores," in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 108–118. [Online]. Available: https://doi.org/10.1145/3372799.3394369

[95] J. J. Grefenstette, "Optimization of control parameters for genetic algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 16, no. 1, pp. 122–128, 1 1986.

[96] T. Back, U. Hammel, and H. . Schwefel, "Evolutionary computation: comments on the history and current state," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 3–17, April 1997.

[97] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," 1977.

[98] "NVIDIA Jetson TX2 Module." [Online]. Available: https://developer.nvidia.com/embedded/jetson-tx2

[99] M. R. Soliman, G. Gracioli, R. Tabish, R. Pellizzoni, and M. Caccamo, "Segment streaming for the three-phase execution model: Design and implementation," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 260–273.