2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)

A memory-centric approach to enable timing-predictability within embedded many-core accelerators

Paolo Burgio*, Andrea Marongiu† Paolo Valente*, Marko Bertogna*
*University of Modena and Reggio Emilia, Modena, Italy
†Swiss Federal Institute of Technology (ETH), Zurich, Switzerland
Email: {paolo.burgio, paolo.valente, marko.bertogna}@unimore.it, a.marongiu@iis.ee.ethz.ch

Abstract

There is an increasing interest among real-time systems architects for multi- and many-core accelerated platforms. The main obstacle towards the adoption of such devices within industrial settings is related to the difficulties in tightly estimating the multiple interferences that may arise among the parallel components of the system. This in particular concerns concurrent accesses to shared memory and communication resources. Existing worst-case execution time analyses are extremely pessimistic, especially when adopted for systems composed of hundreds-to-thousands of cores. This significantly limits the potential for the adoption of these platforms in real-time systems. In this paper, we study how the predictable execution model (PREM), a memory-aware approach to enable timing-predictability in real-time systems, can be successfully adopted on multi- and many-core heterogeneous platforms. Using a state-of-the-art multi-core platform as a testbed, we validate that it is possible to obtain an order-of-magnitude improvement in the WCET bounds of parallel applications, if data movements are adequately orchestrated in accordance with PREM. We identify which system parameters mostly affect the tremendous performance opportunities offered by this approach, both on average and in the worst case, moving the first step towards predictable many-core systems.

1. Introduction

In the last decade, embedded systems embraced heterogeneous designs, where a powerful, general-purpose *host* processor is coupled to massively parallel *accelerators* featuring hundreds of simple and energy-efficient cores, grouped into *clusters* to achieve architectural scalability [1], [2], [3], [4]. Figure 1 shows a possible system-on-chip (SoC) following this template. Such a technology is potentially mature enough for adoption also in the real-time domain, but, unfortunately, current techniques for timing analysis are not effective when applied to the complex hierarchical memory system of modern many-cores. The reason is that classic real-time theory usually views memory latencies as implicit components of the worst-case execution time of tasks, and the interference among cores concurrently accessing memory is upper-bounded to provide a safe worst-case analysis. When moving to multi- and many-cores, the number of processors sharing common memory banks increases, leading to a significantly higher memory contention and more pessimistic worst-case upper bounds.

In this paper, we would like to show that *it is possible to reduce and/or more tightly upper-bound the duration of memory contentions* by using recently proposed memory-aware execution models, enhancing the predictability of multi-core real-time systems. In the PRedictable Execution Model (PREM) [5], code and data are moved prior to execution (*prefetched*) in a private core resource (in that case, the L1 cache), removing the sources of contention at run time, and ultimately reducing WCET bounds.

We believe that the memory hierarchies of modern many-cores can further boost the effectiveness of the PREM model, and, in general, improve WCET predictability. The reason is that, in order to match the stringent energy/area constraints of modern embedded systems, platform architects partially or totally replace "traditional" data caches with explicitly managed memories such as scratchpads (SPMs), which may consume up to 40% less energy and occupy up to 34% less area than caches [6], [7], [8]. The key point is that *the behavior of explicitly-managed scratchpads is also much more predictable than that of caches*, because access latencies are independent of the access pattern [9], [10], [11], and this potentially enhances the benefits of the PREM model, when applied to them.

In this paper, we apply the PREM model to heterogeneous multi-/many-core embedded platforms with explicitly managed memories, enabling a predictable exploitation of the tremendous performance potential of these promising devices. The main novelty of our approach stems from the differences between the targeted platform and traditional multi-cores [5]. We aim at assessing the applicability of PREM on a generic heterogeneous (accelerator-based) embedded system, without requiring specific hardware such as memory shapers, bus bridges, or prefetch instructions that may not be available in the target platform [5], [12].

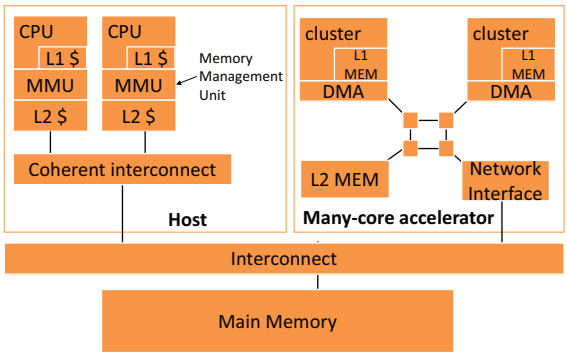


Figure 1: Heterogeneous embedded SoC template

This work focuses on the accelerator engine (rightmost part of Figure 1), which has a significantly different design than the (host) architectures for which the PREM model was first

978-1-4673-8047-8/15/$31.00

proposed. The (many) cores in the accelerator are designed for energy efficiency, and are thus based on a simpler instruction-set architecture (ISA). Typically, they are not capable of running a full-fledged operating system (like the host counterparts), but usually rely on streamlined middleware or native runtime systems running on top of bare metal [2], [3], [13].

Using a representative embedded heterogeneous system, namely, the TI Keystone II EVMK2H board [14], we demonstrate the great potential of the PREM execution model for achieving predictable execution times on embedded multi-/many-core platforms. We model the principal system-level components that influence the execution time of parallel tasks, identifying how the WCET varies depending on the number of cores. These results constitute a first step towards the definition of the necessary models and system background to develop sound memory-aware scheduling algorithms and schedulability tests for heterogeneous many-core systems based on PREM.

This paper is structured as follows. Section 2 reviews a few works related to our contribution, and describes the PRedictable Execution Model. Section 3 shows the heterogeneous architecture considered in this work, and the testbed platform, while Section 4 introduces our timing analysis methodology for PREM and non-PREM applications running on top of it. Experiments are presented in Section 5 to validate the approach. Finally, Section 6 concludes the paper.

2. State of the art and overview of PREM

In the recent years, an increasing number of works explicitly modeled memory delays in the schedulability analysis of multi-core systems. The memory access latency is basically split in two parts: i) the time that a task must wait due to simultaneous concurrent memory accesses (*memory-contention delay*), and ii) the time required to physically retrieve or store a datum (*physical-access delay*).

Classic real-time theory usually views the above latencies as implicit components of the worst-case execution time of tasks. The size of these components is then typically estimated under the assumption that task scheduling algorithms are unaware of the relation between the schedule they produce and the resulting memory latencies. Unfortunately, this approach becomes more and more pessimistic as the number of cores sharing common memory areas increases. Whereas, on one side, memory contention increases with this number, on the other side, with appropriate memory-aware solutions, it is possible to reduce or upper-bound more tightly the duration of memory contentions. This is one of the main motivations of our proposal and of the literature surveyed in this section.

Solutions for controlling memory contention. In [12], Pellizzoni et al. analyze the impact of commercial off-the-shelf (COTS) peripherals on the task execution times for single processor systems, providing a solution for preserving real-time feasibility in the presence of heavy I/O. Peripherals with a heavy I/O load are shown to increase the overall memory access latency, making jobs last up to 44% longer. A special *peripheral gate* is proposed to shape the traffic coming from external devices for preserving the schedulability of the real-time tasks.

Yun et al. [15] introduced *MemGuard*, a framework that aims at guaranteeing memory performance isolation similarly to what happens for CPUs. Each core is assigned a fraction of the memory bandwidth, controlling per-task memory access rates to obtain a lower memory worst-case delay. In [16], the same authors exploited the above memory-throttling strategy for dealing with mixed-criticality fixed-priority tasks, where one core is dedicated to execute critical tasks, while the remaining cores execute best-effort workload.

Explicitly-managed local memories. Explicitly-managed local memories are a well known paradigm for real-time systems, as well as for general multi- and many-core systems. They may consume up to 40% less energy and occupy up to 34% less area than caches [6], [7], [8]. They are also cheaper than caches. Finally, and most importantly for real-time applications, *their behavior is much more predictable*, as access latencies are independent of the access pattern. A lot of research has been carried out on these memory architectures. A detailed survey is available in [6].

Chattopadhyay et al. [8] statically allocate data into the virtual SPM space, of a multi-core system with no caches, ultimately aiming at increasing the predictability of real-time tasks. Although we don't specifically target systems with no caches (but rather use SPMs to exclusively store per-task input and output data sets), our work proves how approaches such as in [8] can be easily adopted, and are even more effective, if applications follow a PREM-like scheme.

Unfortunately, the use of explicitly-managed local memories entails some unavoidable issues, related to moving data back and forth between a limited-size local memory and a larger shared memory. Examples are pointer aliasing and pointer invalidation. In this respect, dedicated memory management units are proposed in [11], [17] to address these issues without requiring whole-program pointer analysis, while making memory accesses time-predictable.

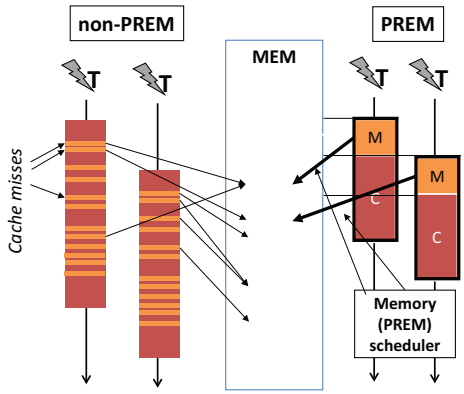


Figure 2: PRedictable Execution Model in a parallel environment

Prefetch-based solutions and PREM. Starting from the seminal work in [12], in [5] Pellizzoni et al. introduced PREM (*PRedictable Execution Model*), a new task execution model in which tasks are split into pairs of *memory* and *computational* phases. Figure 2 shows the distribution of memory accesses both in PREM and non-PREM models. In a first memory phase, tasks retrieve and copy data from the main memory into the local cache of the core they are executing on, whereas, in the following, computational phase, they elaborate non-preemptively previously-cached data. This execution model allows the variability of memory-contention latencies to be greatly reduced, by explicitly controlling memory accesses during memory phases. As such, it allows the overall task execution times to become much more

predictable. Addressing single-core systems, a PREM-compliant co-scheduler is proposed granting main-memory access only when the task being executed on the processor is in the computational phase, without incurring memory conflicts. To enforce this scheduling policy, the co-scheduler relies on the presence of a *Real-Time Bridge*, which arbitrates the access to memory in a time-sharing fashion. This is however **not** the case of our work, whose approach is *completely on the software point of view*, i.e., it does not need additional hardware other than the one which is usually already shipped embedded in a board (e.g., one or more DMA engines).

Different memory-aware scheduling policies for PREM-compliant tasks are evaluated in [18] by simulating synthetic task systems on platform with 4 cores. While failing to detect the worst-case scenario, the simulations show that, on average, the best results are obtained by promoting memory phases over computational phases, *highlighting once more the importance of memory-centric scheduling in multi-core systems*. The schedulability analysis of one of these schedulers is presented in [19].

3. Heterogeneous target platform

This work explores the applicability of the PREM model to heterogeneous multi- and many-core platforms with explicitly-managed shared memories. Figure 1 shows the *generic* architecture targeted in this work. It couples a powerful general-purpose processor (the *host*), featuring sophisticated cache hierarchy, Memory Management Units (MMUs) to support virtual memory and full-fledged operating system, and a programmable manycore *accelerator* composed of one or more clusters of simple processors, to which critical portions (*kernels*) of applications are offloaded.

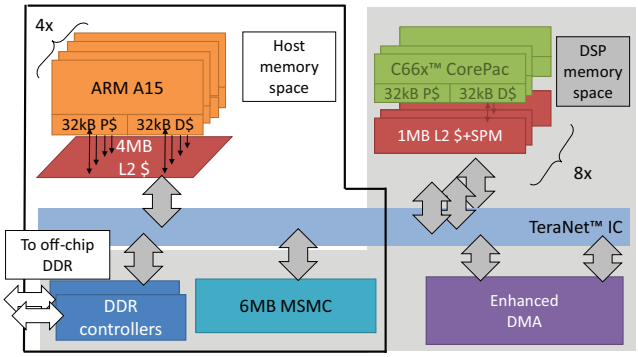


Figure 3: TI Keystone II 66AK2H12 platform.

Keystone II Architecture. As a testbed, we chose the Keystone II™ [14] by Texas Instruments, a widely-known, well supported multi-core platform available on the market. Figure 3 shows the target board, named Keystone II EVMK2H [14], which embeds an ARM ®Cortex host Quad-core and a cluster of eight Digital Signal Processing cores of the TMS320C66x family (also alled *CorePac*) as an accelerator. While the final target of our work are many-core systems composed of tens-to-hundreds of cores, the adopted platform allows modeling typical design choices made at cluster level for cluster-based many-core systems such as [1], [2], [3], [4]. In this sense, the accelerator subsystem of Keystone II can be seen as a single cluster of a many-core design. This work focuses on the implementation of the PREM execution model at cluster level. Introducing multiple clusters in the model will be the next step of our research.

Keystone II memory system and PREM. In TI Keystone II, each DSP is a Very-Long Instruction Word core (VLIW), with L1 and L2 caches, and a L2 software-managed data memory, which we will leverage to implement PREM. The platform has also an on-chip scratchpad memory shared among the host and the accelerator, called Multicore Shared Memory Controller, or MSMC, and an off-chip DDR SRAM. Figure 3 also highlights the host and accelerator memory space (respectively the boxed and gray areas): due to these memory space restrictions, communication between the host and the accelerator sub-system happens **only** through the shared banks of the MSMC and DDR memory. For this reason, under PREM, the full working data set is atomically moved off and forth the MSMC and the local L2 scratchpad to implement the memory – M phase. This can be done in a very efficient way using the on-chip DMA engine, called Enhanced Direct Memory Access (EDMA). Since the DMA is a unique shared resource, concurrent M phases will be sequentialized, introducing a potential bottleneck that we aim at analyzing.

Programming the Keystone II. Heterogeneous many-cores are significantly different from "traditional" single-core architectures, mainly for their complex, hierarchical memory system. As a consequence, programming models have evolved through the years to include heterogeneous computing resources (host vs. accelerators) and to expose the memory system and data transfers to programmers. Noticeable examples are OpenCL [20] and the recent OpenMP 4.0 specifications [21], both supported in Keystone II. Currently, software architects rely on the so-called *offload* execution model, where an application runs on the host cores, and its computationally-intensive kernels are executed in the accelerator subsystem. After the offloading sequence, parallel threads running on the cores of the accelerator subsystem are responsible for triggering additional DMA transfers into local or private memories (see Figure 5), if needed. We organize these transfers to implement the memory – M phase of PREM tasks.

4. Architecture modeling and Worst Case analysis

In this section we explore the applicability of PREM to the target multi-core platform. The advantage of PREM-compliant code against "traditional" code is that all processing happens on local data, with no interfering traffic [1]. Hence, the worst-case timing analysis can be less "conservative" than in the non-PREM case, leading to lower WCET estimates. In traditional analysis made on non-PREM code, we don't "pay the price" of the M phase and its DMA transfers, but we then must take into account the contention for accessing the (shared) MSMC banks by the parallel tasks.

We first describe the timing model of the platform, which we then use to support the worst-case execution time (WCET) analysis of a generic application running on it. Table 3 summarizes the main architectural parameters of the Keystone II 66AK2H12 board [14]. Latency estimates for MSMC and L2 (two lowermost rows) come from Texas Instruments [2]. Here, it is important to recall that the multi- and many-core accelerators we consider are

1. Several techniques exist to identify the "local data" of a task, either *via* compiler analysis [5], or making them explicit using programming models such as OpenCL [20] or OpenMP [21]. However, we do not cover this aspect, here.

2. For more details, see official Keystone II tutorials by TI, e.g., http://keystone-workshop.googlecode.com/svn/trunk/preliminary/MoreAboutCache_AndMPAX.pptx

significantly simpler than a "standard" core such as the one in the host (see Sections 1 and 3). Since these cores typically don't feature, e.g., MMUs nor branch predictors, the complexity of worst-case analysis is greatly reduced, and this is a key point of our approach.

NOP instr.	*Specified*	Parallel instr.	0 cyc.
Normal instr. (t_{asm_instr})	1 cyc.	LD (cached)	+0 cyc.
LD – Load (MSMC) (t_{LD})	+20 cyc.	LD (L2) (t_{LD})	+7 cyc.
ST – Store (MSMC)	+0 cyc.	ST (L2)	+0 cyc.

Table 1: C66x instruction costs.

Modeling the 'C' phase of PREM, and non-PREM code. Parameters in Table 3 allow us to model application execution time on the Keystone II. Being C66x a VLIW architecture with a width of 8 instructions, its CPI (Cycles-per-instruction) varies from 0,125 ($= 1/8$, in case 8 instruction are executed in parallel) to 1 (when only 1 instruction is executed). We developed a tool which analyzes the assembly code produced by the C66x™ compiler (from Texas Instrument's MCSDK), and assigns a "penalty" for each instruction.

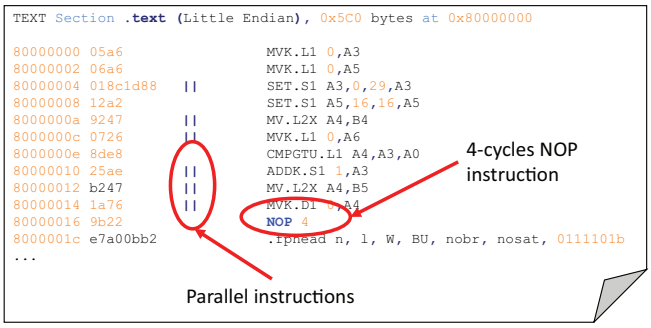


Figure 4: Excerpt of C66x code.

Figure 4 shows an excerpt of C66x assembly code. As shown, parallel instructions are marked with the '||' symbol, while a NOP instruction has an argument representing the number of stall cycles of the core. We perform a WCET analysis of both i) the non-PREM version and ii) the computation phase (C) of the PREM version of an application, by assigning a timing cost to each instruction, depending on its type. Table 1 shows the costs considered for every type of instruction. With these numbers, we can simply derive the time spent in the computing phase as:

$$t_C = \sum_{\forall \text{instruction kinds}} n_{asm\_instr} * t_{asm\_instr}, \qquad (1)$$

where n_{asm_instr} and t_{asm_instr} represent the number of instructions of a particular kind, and the time it takes to execute the instruction, respectively.

For load (LD) and store (ST) instructions, Table 1 shows *additional* cycles (other than the ones already spent in the core pipeline). The extra cost for a memory load t_{LD} can be either 20 or 7 cycles depending on where data is allocated (MSMC or L2): t_{asm_instr} for LD and ST are increased with numbers from Table 1, depending on the memory bank where related data are. The L1 data cache has a write-through policy, but the cost for a write is 0 cycles thanks to the use of asynchronous post-storing techniques (a.k.a. *delayed-storing*, or *lazy-storing*). Putting these numbers in Equation 1, we correctly capture the execution time of both non-PREM applications (data resides in MSMC) and PREM applications (data resides in local L2 memory) without accounting for the M phase.

Block size	1 block	2 blocks	4 blocks	8 blocks
4 kB block	0.488	0.366	0.427	0.336
32 kB block	0.153	0.137	0.130	0.130
64 kB block	0.122	0.114	0.114	0.111

Table 2: DMA cost for transfering a single byte (microseconds)

Modeling the 'M' phase of PREM. In order to perform WCET analysis of PREM-compliant applications, we also need to correctly model the DMA transfers from global MSMC to local (L2) scratchpad memories. To do so, we performed an extensive set of experiments to derive the minimum bandwidth for DMA transfers of blocks of different sizes. For reasons of space we will not describe our analysis in details here, but only the main outcomes. In a typical scenario for the target system [13], [20], [21], [22], the M phase moves one or more contiguous blocks of data off and forth the accelerator subsystem. We found that, for block sizes of more than 32kB, the DMA programming cost gets amortized, and performance scale linearly (i.e., transferring twice the data takes twice the time). Table 2 shows the estimate of the cost in microseconds (t_{Byte}) for transferring a single Byte of data, and with this we can estimate the worst-case time for a DMA transfer (the M phase) as:

$$t_M = \sum_{i \in input\_vars} [size\_Bytes_i * t_{Byte} * n_{threads}], \qquad (2)$$

where $n_{threads}$ represents the number of threads simultaneously competing for memory resources. Note that if the core initiating the transfer runs in isolation, then $n_{threads} \equiv 1$, that is, there is no interfering traffic towards the memory system.

Since instruction caches are warmed up before execution, we may neglect the impact of instruction cache misses, such as, e.g., Puauat et al. do in [23].

5. Experimental validation

In this section, we validate our approach on the Keystone II, comparing PREM and non-PREM versions of the same benchmark. At first, we introduce and discuss a synthetic benchmark we developed that stresses the core and memory system, and run it on an varying number of cores to extract the execution time, both for the PREM and non-PREM versions. As explained in Section 3, in the Keystone II platform, data are placed by default in the MSMC. In the PREM version, we prefetch them in the per-DSP local L2 SPM; in the non-PREM version of the benchmark, data are left in the (cacheable) MSMC. In a second set of experiment, we test the accuracy of our timing model, i.e., we compare the results obtained for the PREM version against the prediction of the analysis based on the platform characterization presented in Section 4. We then study the performance of a representative application – a matrix multiplication – written in OpenCL [20] for the target platform. Finally, we explore how application performance is affected by their computation-to-communication ratio, that is, the percentage of time which is spent on the memory – M phase.

Core frequency	1GHz	Cycles-per-instruction (CPI)	0.125-1
L1 D$ size	32kB	L2 mem size	256kB
L2 mem write latency(cyc)	0	L2 SPM read latency (cyc)	7
MSMC size	1MB	MSMC latency (cyc)	20

Table 3: Keystone II experimental setup.

# Cores/threads	1	2	4	8
No-PREM – Worst (Analytical)	0.026	0.047	0.088	0.170
PREM – Worst (Analytical)	0.010	0.014	0.022	0.038
Speedup	2.6×	3.4×	4.0×	4.5×

Table 4: Analysis of the WCET of synthetic benchmark (microsec.)

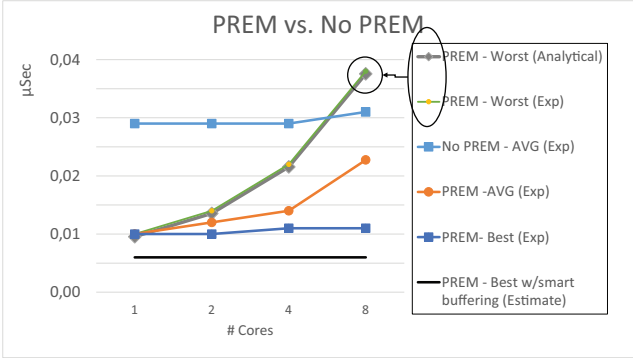
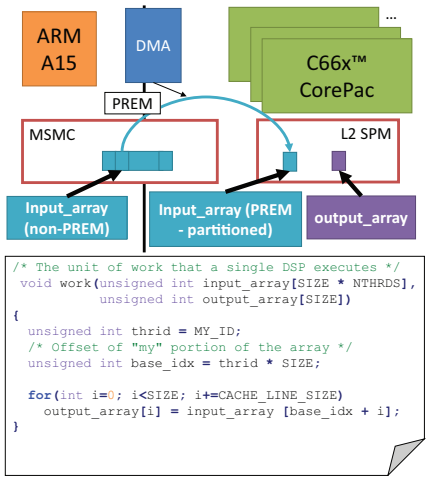


Figure 6: Synthetic benchmark execution times (both Analytical and Experimental)

5.1.Synthetic Benchmark description

We developed a synthetic benchmark which stresses the data memory by accessing an array in such a way it continuously causes cache misses. Figure 5 describes the job that is performed on each DSP. There is a global (shared) `input_array` which is




```
/* The unit of work that a single DSP executes */
void work(unsigned int input_array[SIZE * NTHRDS],
          unsigned int output_array[SIZE])
{
  unsigned int thrid = MY_ID;
  /* Offset of "my" portion of the array */
  unsigned int base_idx = thrid * SIZE;

  for(int i=0; i<SIZE; i+=CACHE_LINE_SIZE)
    output_array[i] = input_array [base_idx + i];
}
```

Figure 5: Synthetic benchmark.

split in `NUM_THREADS` parts of `SIZE` elements, and each thread copies (item-by-item) it into another array. In our experiments, *SIZE* is 32kB.

In a default program deployment scheme on the Keystone, `input_array` resides in the MSMC, because the benchmark starts running on the host, which has no access to the local (L2 and L1) DSP memories. As explained in Section 3, this is due to memory space restrictions. With PREM, we move data to the L2 SPM by means of DMA transfers, and computation is performed locally. For the moment, we do not consider the writeback phase, i.e., `output_array` always resides in the local L2 memory of the accelerator. The model can easily be enhanced to deal with it, as Alhammad et al. show in [24].

5.2.Performance analysis of the synthetic benchmark

Table 4 shows the estimated performance comparison of PREM and non-PREM version of the synthetic benchmark, with the number of threads varying from 1 to 8 (each threads runs on a dedicated DSP core). It also shows how performance of PREM scales better with the number of threads. Times are in microseconds, and correspond to the WCET as derived from our analytical model in Section 4. They show that PREM outperforms non-PREM by more than $4\times$, for 8 parallel threads. This impressive result mainly descends from the tremendous memory bandwidth of DMA burst transfers (128B wide bursts) under PREM, when compared to the cache fill time (20 cycles for a 64B line) of the non-PREM model. This is particularly important in a typical real-time scenario, where parallel threads have different (fixed) priorities in accessing shared resources. Here, the worst-case performance of the PREM model corresponds to the **lowest priority thread** that suffers full memory interference from the memory phases of all other threads. With a memory-aware task mapping and scheduling algorithm in place[3], it would be then possible to select which task to assign to this "unlucky" thread, reserving the **higher priority threads** for more timing sensitive tasks.

This is shown in the bottom lines of Figure 6: the line *"PREM-Best"* shows that under PREM, the highest priority thread never encounters memory interference, hence has a significantly smaller execution time than the low-priority "unlucky" thread previously mentioned, and that his performance do not significantly change when the number of concurrent cores increases.

The most interesting result in Figure 6, however, is that the two topmost curves related to the prediction of our PREM analytical model and the actual results from experiments (*"PREM-Worst (Analytical)"* and *"PREM-Worst (Exp)"*) overlap almost perfectly. This means that our model is capable of correctly capturing the behavior of the synthetic benchmark, which, as explained, was explicitly written to simulate a scenario of high traffic towards the accelerator memory system. Table 5 shows

3. Designing a memory-aware scheduler is not the purpose of this work, nor it is dealing with recurring task models. However, this work indeed aims at providing the necessary system-level background to design such scheduling strategies as a next step of our research.

# Cores	1	2	4	8
Worst (Analytical)	0.009	0.013	0.021	0.037
Worst (Experimental)	0.010	0.014	0.022	0.038

Table 5: Synthetic bench: accuracy of the analytical model (microsec).

the absolute times (in microseconds), which are almost identical, besides an acceptable error due to measuring resolution and approximation. We show results for *"no-PREM - Worst"* only in Table 4 and not in Figure 6 because they would harness the scale – hence the readability – of the plot.

Figure 6 also shows the experimental results for the average (*AVG*) execution times i.e., the sum of the experimentally measured execution times of all threads divided by the number of threads. Somewhat surprisingly, also these results show the PREM curves outperforming the non-PREM ones, even if by a smaller factor that decreases with the number of threads.

Boosting up the performance of PREM. A potential issue in any prefetch-based technique is that, if explicit (DMA) data transfers are not carefully dimensioned and managed, one may end up in prefetching a larger amount of data than needed. This potentially causes an increase in the average execution time due to the initial latency of the memory phase, especially when the number of threads increases (remember that concurrent memory phases are sequentialized, under PREM). However, this effect can be significantly reduced by means of double buffering techniques [25], which enables overlapping of memory – M and computation – C phases of two consecutive tasks. When the first memory phase finishes, the running task can immediately start the computational phase, while the next task starts its memory phase, and so on. If these M–C phases are properly dimensioned, and if the application allows it, it is therefore possible to "hide" all of the data accesses. The bottom black line in Figure 6 (*"PREM - Best w/smart buffering"*) shows the performance *estimate* of the highest-priority thread, when memory latencies of the M phase are completely hidden using smart buffering techniques. The difference between the two lowermost lines shows how being able to "hide" data transfers, e.g., with double buffering, potentially further improves performance, here by a factor of $\approx 2\times$, even for "privileged", high-priority threads, and does not vary increasing the number of threads.

5.3.Matrix multiplication

We choose the matrix multiplication as a "real" application testbed for our approach. We modified according to PREM the code of the *matmpy* benchmark that comes with the Keystone SDK provided by Texas Instruments. This code is written in OpenCL [20]. In this application, one ARM core decomposes the output matrix into sub-matrices, and offloads the computation of the sub-matrices to the DSP cores. We use the full set of 8 DSPs (8 parallel threads). Figure 7 shows the partitioning of application in small identical parallel tasks, which in OpenCL are called *work-items*.

The version of the application written by TI employs data prefetching to boost performance: input data is moved into the L2 memory, but in a non-PREM fashion, that is, the access to the (shared) DMA resource is not mediated, and parallel threads concurrently access it. We thus explore three different versions of the application:

1) NO-PREFETCH – a "naive" version with no prefetching, where the three matrices reside in MSMC (which is the default placement of the OpenCL data buffers in Keystone II);
2) PREFETCH – the "original" version, where prefetch is used to increase data locality, but not following the PREM model, and
3) PREM – a version which follows the PREM model, i.e., where the parallel threads lock the DMA resource during the whole M phase, which is performed at the beginning of each OpenCL work-item.

Figure 8 shows the performance (in seconds) an OpenCL work-item for each of the three versions, when run on Keystone II. It shows performance for the average, best (highest priority thread) and worst (lowest priority thread) case. Results show that prefetching data improves performance by $\approx 8\times$, and PREM gives an additional $\approx +20\%$ improvement. The excellent performance gain of the prefetch-based versions is due to the reduction of cache trashing: each matrix occupies 256kB, while L1 data caches of DSP cores are only 32kB, causing a high number of cache misses. By increasing data locality in the prefetch version (hence, also under PREM) we mitigate this effect. We easily demonstrate this re-running the benchmark with matrices of size down to 4kB, which entirely fit in the cache. Figure 10 shows that no-prefetch and prefetch versions perform almost identically for matrices smaller than 64kB.

Figure 9 shows our WCET prediction for *matmpy*, and the corresponding experimental values. WCET is improved by a factor of $\approx 2\times$ using prefetch, and that PREM outperforms the prefetch-based version by $\approx 3\times$. Thorough readers might have noted than the difference between the prediction and the actual (*EXPerimental*) numbers is higher than for the synthetic benchmark. The reason is that, *matmpy* is not explicitly designed to stress the memory system, hence the WCET is not likely to appear even with a high number of experiments, such as it happens with the synthetic benchmark [4].

5.4.Performance as a function of Memory Ratio

In this section, we show the **expected** worst-case performance on the target platform for generic PREM-compliant applications characterized by a given *memory ratio (MR)*, defined as the total number of cycles spent in the M phase, divided by the total number of cycles of the application, when run in isolation:

$$MR_{PREM} = \frac{t_M}{(t_M + t_C)} \tag{3}$$

Besides the application code itself, the memory ratio of a PREM-compliant application depends of multiple factors, such as memory latencies, DMA bandwidth, type of processor, compiler optimizations, etc.

The C66x core has a 10-stage pipeline, where a memory operation takes at least 4 cycles to execute (memory access results are fetched 4 stages "after" the address is emitted). In case there is not enough instruction-level parallelism to exploit in the source code, the compiler inserts 4 NOPs to meet this constraint. Hence, considering a 7-cycles latency for the L2 memory (see Table 1), under PREM each DSP has a maximum MR of:

4. Several existing tools could be used that adopt the "Implicit Path Enumeration Technique" [26]. Note that, also profile-based and probabilistic approaches [27] are recently gaining attention and trust in soft real-time domains, but this is an orthogonal (yet, nowadays, "hot") research direction, and it's out of the scope of this paper.

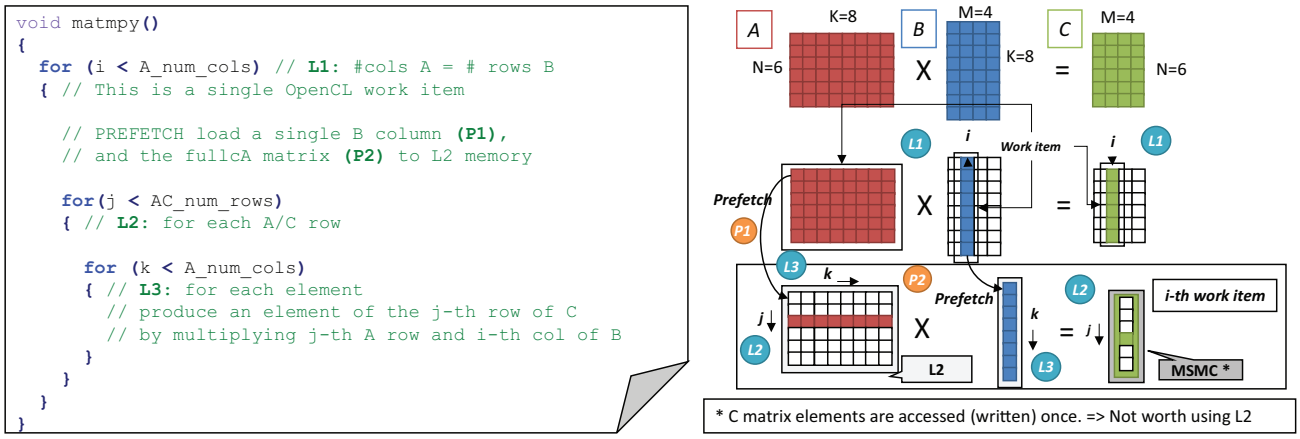


Figure 7: OpenCL matmpy partitioning.

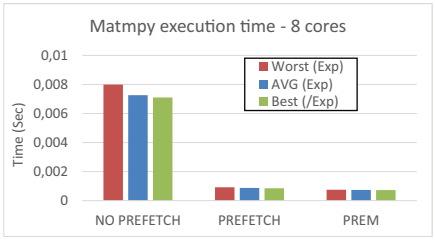


Figure 8: Matmpy: no-prefetch vs. prefetch vs. PREM.

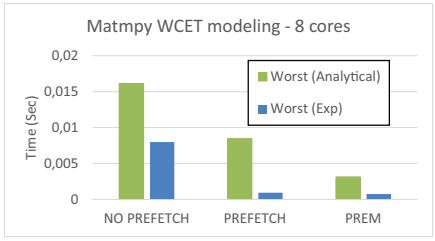


Figure 9: Matmpy performance analytical model.

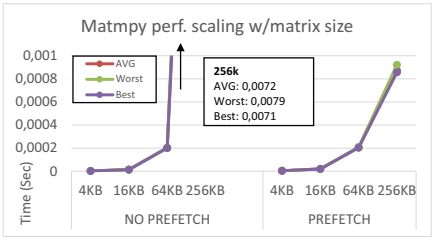


Figure 10: Performance scaling w/ matrix size.

$$MR_{KeystoneII} = \frac{7}{(7+4)} = 0.63 \quad (4)$$

According to our profiling tool, the considered synthetic benchmark has a memory ratio MR_{sbench} of 0.44, and *matmpy* has a memory ratio MR_{matmpy} of 0.56.

Figure 11 shows how the PREM performance deteriorates for different memory ratios and an increasing number of threads, according to our timing model. All times are normalized to the case where a single DSP runs in isolation. From the bottom to the top, we study performance degradation from a minimum *MR* of 0.1 (out of 10 execution cycles, 1 is spent accessing the memory) representing a computational-intensive application, to the maximum achievable on Keystone II ($MR_{Keystone} = 0.63$).

Increasing the number of cores has a different impact on the WCET depending on the memory ratio. In particular, an increment in the WCET of $\sim 2\times$ and $\sim 5\times$ is predicted for the most computational-intensive and memory-intensive cases, respectively, with eight DSP cores.

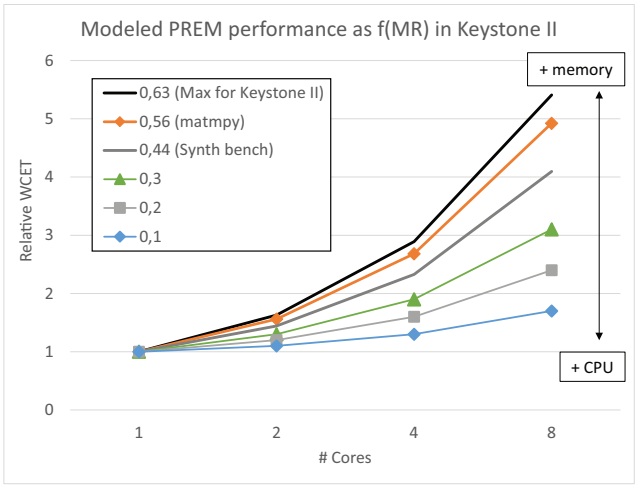


Figure 11: Modeled PREM performance in the Keystone II.

6. Conclusions and future work

Future computing systems will run on heterogeneous platforms, featuring many-core accelerators. Unfortunately, the complexity of worst-case timing analysis on these architectures prevents their adoption in the real-time domain due to pessimistic and over-conservative WCET predictions.

This paper explored the applicability of the PRedictable Execution Model (PREM) to embedded many-core accelerators with explicitly shared memories. This paper shows that PREM is effective in guaranteeing timing predictability in a representative embedded heterogeneous platform, with an order-of-magnitude speedup compared to classical, non-PREM approaches. We modeled the main components that influence the worst-case execution times of the parallel tasks running on the considered multi-core system, and estimate how the performance varies depending on the number of cores and the memory to computation ratio. Results validate the effectiveness of the PREM execution model when adopted on a representative multi-core architecture, as well as the accuracy of the proposed timing model. This work provides us with the basics and necessary model to develop smart scheduling strategies for PREM-compliant applications executing upon next-

generation heterogeneous systems.
As a next step, we intend to consider more complex application, i.e., with multiple recurring real-time tasks, devising smart (memory and CPU) scheduling strategies, and associated schedulability analysis, to maximize the feasibility. We would also like to investigate the optimal resource allocation and dimension of local buffers, and the tradeoff/granularity between M and C phases, for instance identifying *classes* of applications, starting from the achievements in Section 5.4, and exploring their behavior under PREM. Finally, we want to generalize the presented technique to hierarchically scheduled memory transfers for cluster-based architectures, enriching our model by considering multiple DMA modules, and multiple clusters which sum up to hundreds/thousands of cores. We believe these will soon be hot research topics in the real-time and embedded systems domains.

Acknowledgments

This work has been supported in part by the European Commission under the P-SOCRATES project (FP7-ICT-611016).

References

[1] Adapteva, Inc., "Epiphany-IV 64-core 28nm Microprocessor," [Online] http://www.adapteva.com/products/silicon-devices/e64g401/, 2013.

[2] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, 2012, pp. 983–987.

[3] Kalray Corporation, "Many-core Kalray MPPA," [Online] http://www.kalray.eu/, 2012.

[4] NVIDIA, "Next Generation CUDA Compute Architecture: Fermi - WhitePaper," [Online] http://www.nvidia.fr/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2010.

[5] 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 *Proceedings of the 17th IEEE International Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS '11, April 2011, pp. 269–279.

[6] A. Marongiu and L. Benini, "An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs," *Computers, IEEE Transactions on*, vol. 61, no. 2, pp. 222–236, Feb 2012.

[7] L. Karam, I. AlKamal, A. Gatherer, G. Frantz, D. Anderson, and B. Evans, "Trends in multicore DSP platforms," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 38–49, November 2009.

[8] S. Chattopadhyay and A. Roychoudhury, "Static bus schedule aware scratchpad allocation in multiprocessors," *SIGPLAN Not.*, vol. 46, no. 5, pp. 11–20, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/2016603.1967680

[9] I. Puaut and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison," in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, April 2007, pp. 1–6.

[10] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable programming on a precision timed architecture," in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '08. New York, NY, USA: ACM, 2008, pp. 137–146. [Online]. Available: http://doi.acm.org/10.1145/1450095.1450117

[11] S. Wasly and R. Pellizzoni, "A dynamic scratchpad memory unit for predictable real-time embedded systems," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, July 2013.

[12] R. Pellizzoni, B. Bui, M. Caccamo, and L. Sha, "Coscheduling of CPU and I/O transactions in COTS-based embedded systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, Nov 2008, pp. 221–231.

[13] A. Marongiu, P. Burgio, and L. Benini, "Fast and lightweight support for nested parallelism on cluster-based embedded many-cores," in *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*, 2012, pp. 105–110.

[14] Texas Instrument Inc., "The 66AK2H12 Keystone II Processor." [Online]. Available: http://www.ti.com/product/66AK2H12

[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 *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 55–64.

[16] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, ser. ECRTS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 299–308.

[17] J. Whitham and N. Audsley, "Implementing time-predictable load and store operations," in *Proceedings of the Seventh ACM International Conference on Embedded Software*, ser. EMSOFT '09. New York, NY, USA: ACM, 2009, pp. 265–274. [Online]. Available: http://doi.acm.org/10.1145/1629335.1629371

[18] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo, "Memory-aware scheduling of multicore task sets for real-time systems," in *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, ser. RTCSA '12, Aug 2012, pp. 300–309.

[19] 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. [Online]. Available: http://dx.doi.org/10.1007/s11241-012-9158-9

[20] Kronos Group, "The OpenCL 1.1 Specifications," 2010. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[21] "OpenMP Application Program Interface v4," 2011. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[22] P. Burgio, G. Tagliavini, A. Marongiu, and L. Benini, "Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, 2013, pp. 1504–1509.

[23] I. Puaut, "WCET-centric software-controlled instruction caches for hard real-time systems," in *Real-Time Systems, 2006. 18th Euromicro Conference on*, 2006, pp. 10 pp.–226.

[24] A. Alhammad and R. Pellizzoni, "Time-predictable execution of multithreaded applications on multicore systems," in *Proceedings of Design, Automation and Test in Europe*, ser. DATE '14, Mar 2014.

[25] J. Sancho and D. Kerbyson, "Analysis of double buffering on two different multicore architectures: Quad-core opteron and the cell-be," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–12.

[26] P. Puschner and A. Schedl, "Computing maximum task execution times – a graph-based approach," *Real-Time Systems*, vol. 13, no. 1, pp. 67–91, 1997.

[27] V. Nélis, P. M. Yomsi, and L. M. Pinho, "Methodologies for the WCET Analysis of Parallel Applications on Many-core Architectures," in *18th Euromicro Conference on Digital System Design, DSD 2015*, August 2015.