Evaluating Controlled Memory Request Injection for Efficient Bandwidth Utilization and Predictable Execution in Heterogeneous SoCs

GIANLUCA BRILLI, Department of 'Ingegneria Enzo Ferrari', University of Modena and Reggio Emilia, Modena, Europe, gianluca.brilli@unimore.it

ROBERTO CAVICCHIOLI, Department of Sciences and Methods for Engineering, University of Modena and Reggio Emilia, Reggio Emilia, Europe, roberto.cavicchioli@unimore.it

MARCO SOLIERI, PAOLO VALENTE, and ANDREA MARONGIU, Department of Physics, Informatics and Mathematics, University of Modena and Reggio Emilia, Modena, Europe, name.surname@unimore.it

High-performance embedded platforms are increasingly adopting heterogeneous systems-on-chip (HeSoC) that couple multi-core CPUs with accelerators such as GPU, FPGA or AI engines. Adopting HeSoCs in the context of real-time workloads is not immediately possible, though, as contention on shared resources like the memory hierarchy – and in particular the main memory (DRAM) – causes unpredictable latency increase. To tackle this problem, both the research community and certification authorities mandate (i) that accesses from parallel threads to the shared system resources (typically, main memory) happen in a mutually exclusive manner by design, or (ii) that per-thread bandwidth regulation is enforced. Such arbitration schemes provide timing guarantees, but make poor use of the memory bandwidth available in a modern HeSoC. *Controlled Memory Request Injection* (CMRI) is a recently-proposed bandwidth limitation concept that builds on top of a mutually-exclusive schedule but still allows the threads currently not entitled to access memory to use as much of the unused bandwidth as possible without losing the timing guarantee. CMRI has been discussed in the context of a multi-core CPU, but the same principle applies also to a more complex system such as an HeSoC. In this paper we introduce two CMRI schemes suitable for HeSoCs: *Voluntary Throttling* via code refactoring and *Bandwidth Regulation* via dynamic throttling. We extensively characterize a proof-of-concept incarnation of both schemes on two HeSoCs: an NVIDIA Tegra TX2 and a Xilinx UltraScale+, highlighting the benefits and the costs of CMRI for synthetic workloads that model worst-case DRAM access. We also test the effectiveness of CMRI with real benchmarks, studying the effect of interference among the host CPU and the accelerators.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; **Real-time systems**; • **Hardware** → *Reconfigurable logic and FPGAs.*

Additional Key Words and Phrases: Heterogeneous systems-on-chip, Memory interference, Predictable Execution

ACM Reference Format:
Gianluca Brilli, Roberto Cavicchioli, Marco Solieri, Paolo Valente, and Andrea Marongiu. 2018. Evaluating Controlled Memory Request Injection for Efficient Bandwidth Utilization and Predictable Execution in Heterogeneous SoCs. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In the last decade high-end embedded systems have successfully embraced parallelism and heterogeneity as key design paradigms to deliver an ever-increasing demand for peak performance within tight power envelopes. Current

Manuscript submitted to ACM

heterogeneous systems-on-a-chip (HeSoC) are composed of several compute units like multi-core CPUs and one or more programmable accelerators (e.g., a GPU) or programmable logic (FPGA). These systems are characterized by high memory contention on shared caches, coherent interconnects, DRAM controller and the DRAM itself, which impacts the latency of memory operations and ultimately the timing of the executing workloads. This happens because main memory accesses are regulated for best-effort performance rather than predictable timing behavior. This has so far prevented the adoption of commercial-off-the-shelf (COTS) heterogeneous platforms for the design of safety-critical solutions.

The problem of memory interference has been extensively studied for multi-core systems [25] and high-performance systems [18, 29]. Focused investigations addressed the characterization of memory interference and its impact on latency [10, 31–33].

Recently, the research forefront has started looking at solutions that tackle the problem at the software level, without the need to alter COTS hardware. Two main approaches have emerged. The first is to mitigate interference by limiting the bandwidth of each actor that accesses memory [22, 30]. The total memory bandwidth varies with the pattern of accesses, as a consequence it is hard to compute individual bandwidth limits that allow a high memory utilization to be reached with every workload. Bandwidth-reclaiming schemes could improve utilization, but their effectiveness might be poor in the presence of quick variations of the bandwidth demands. The second approach aims at guaranteeing mutually exclusive access to memory resources by design, as part of the programming model. The Predictable Execution Model (PREM) is a notable example of such an approach, where task execution is structured as a repeating sequence of *memory phases* (that access the shared memory) and *compute phases* (that only use local caches and registers). By scheduling *memory phases* in a mutually exclusive manner memory contention is avoided. Originally formulated to address concurrent accesses between single-core CPU and devices with direct memory access [24], PREM has been later successfully extended to the case of multi-core CPUs [5, 27] and of HeSoCs [15, 16, 20]. Although effective at guaranteeing predictable timing of memory accesses, PREM-like approaches greatly sacrifice memory bandwidth utilization, as bandwidth in a modern HeSoC is sized to concurrently serve multiple computing units. This important bandwidth loss has motivated further research. Yao et al. [35] proposed a simple variant to the PREM model where a memory-centric scheduler allows a number $k > 1$ of *memory phases* to execute in parallel. They observed that the slowdown is quite low especially when the cores use different memory banks. The granularity of the approach, however, is too coarse and lacks control on bandwidth usage, since $k = 2$ may already cause unacceptable latency degradation. Other approaches try to determine the interference suffered by concurrent mixed-criticality applications with different memory access patterns running on COTS HeSoCs and implement bandwidth control mechanisms accordingly [9] [14]. Regardless of the proposed solution, all these papers show that a high utilization can be reached only by allowing multiple cores to access memory at the same time.

Controlled Memory Request Injection (CMRI) [11] was recently discussed as a novel concept to enabling fine-grained control of the unused bandwidth in PREM-like schemes. CMRI promises better usage of DRAM bandwidth while at the same time maintaining the task under service's latency increase within acceptable bounds. This is achieved by coupling a conservative choice of k active *memory phases*, possibly $k = 1$, with a mechanism for carefully-dosing memory requests injection from other cores.

CMRI has been presented as a conceptual model to be used in the context of a multi-core CPU, but dedicated CMRI schemes could be devised also for more complex systems such as an HeSoC. Here, the sources of interference are more numerous and typically of a greater magnitude – it suffices to compare the peak bandwidth used by a multi-core CPU to that of a GPU on a modern HeSoC (see Table 1 later on). This paper moves the first, fundamental steps towards the

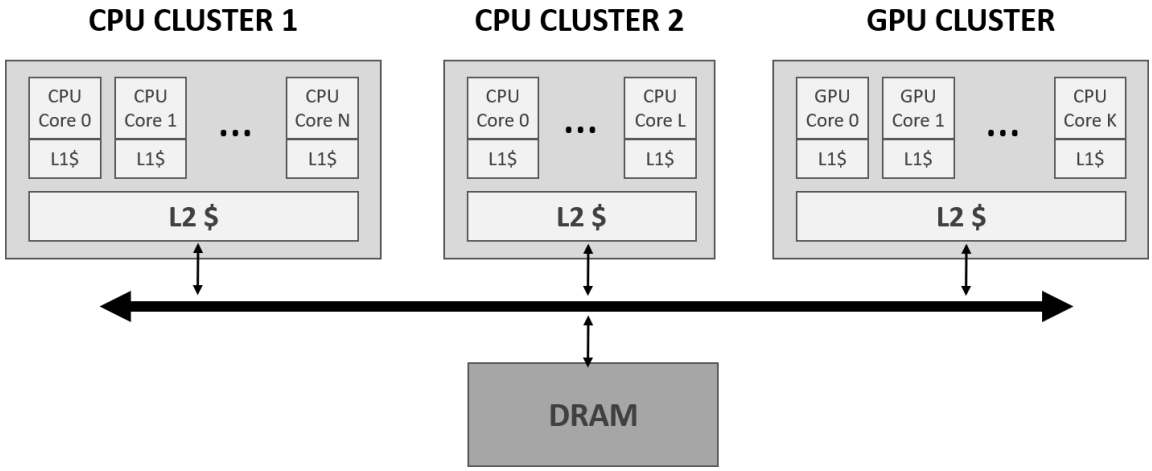

Fig. 1. Cluster-Based HeSoC Architectural Template.

definition and the realization of practical CMRI schemes for modern commercial HeSoCs. Specifically, we make the following contributions: First, we describe two approaches to support CMRI in a HeSoC: *Voluntary Throttling* (VOLT) and *Bandwidth Regulation* (BWR).

- *VOLT* is based on static (compile-time) transformation of PREM *memory phases*, with an injection mechanism that is voluntarily initiated by the task code, providing the most fine-grained control;
- *BWR* relies on OS- or hypervisor-level capability of dynamic bandwidth allocation [37].

Second, we present an extensive characterization of both CMRI schemes, based on a proof-of-concept realization of VOLT and BWR for a modern HeSoC, the Nvidia Jetson TX2. Our analysis shows a detailed profiling of memory bandwidth and latency when CMRI is applied to synthetic benchmarks aimed at modeling realistic worst-case access patterns. VOLT and BWR are also compared in terms of both effectiveness of control and cost (performance overhead).

Third, we conduct a set of experiments with real benchmarks (the Polybench suite [28]) to study the benefits of CMRI in the context of realistic workloads. These experiments specifically look at the effect of interference on the benchmarks from memory traffic generated by the SoC accelerator. To make the results for this last experiment as general and comprehensive as possible, we have considered two different types of HeSoCs: one based on a GPU accelerator (the NVIDIA Jetson TX2) and one based on an FPGA accelerator (the Xilinx Ultrascale+ MPSoC). The proposed methodology, framework and experimental results confirm the feasibility and the effectiveness of CMRI applied to HeSoCs. This paves the way for further research on dedicated techniques for determining the proper CMRI values given a target workload.

The paper is organized as follows. In Section 2 we provide background information concerning the architecture template, the features of modern HeSoCs and PREM. In Section 3 we discuss the concept of CMRI and the two practical implementatations: VOLT and BWR. In Section 4 we describe our experimental setup and the various experiments conducted to assess the benefits and costs of CMRI. In Section 5 we discuss related work, while in Section 6 we provide conclusive remarks and future research directions.

Table 1. Baseline and composite bandwidth on recent SoCs.

SoC/Board	CPU Cluster count	Cores per Cluster	Arch	Core BW [GiB/s]	Total CPU BW [GiB/s]	Total SoC BW [GiB/s]
NVIDIA Jetson Xavier AGX	4	2	Carmel	18	74	137
NVIDIA Jetson TX2	2	2 4	Denver2 A57	9 4.3	18 + 6.5 = 24.5 (Tot)	59.7
NXP i.MX 8QM	2	2 4	A72 A57	3 1.2	10.8	23.8
Xilinx UltraScale Zynq ZCU 102	1	4	A53	2.5	7.2	16.5

2 BACKGROUND

Most modern multi-core SoC designs at every scale [3] [13] leverage heterogeneity at different levels. Figure 1 shows the typical architectural template for this type of systems. Usually, a powerful multi-core, general-purpose CPU, the *host* processor, is coupled to some type of acceleration fabric, like a data-parallel co-processor (e.g., a GPGPU). For energy efficiency, the design of the *host* CPU itself is typically heterogeneous, with a number of *compute clusters* locally grouping a small number of homogeneous CPUs sharing interconnection and memory resources (L2 cache). The *cluster* design paradigm is also adopted for programmable accelerators, as it enables scalability. Globally, these heterogeneous *clusters* share the main system memory (DRAM). Table 1 lists the architectural details of some widely adopted commercial products that adhere to the cluster-based heterogeneous design paradigm. In all these systems the main memory must typically sustain tremendous bandwidths to satisfy requests coming from many actors. Overall, the system is designed for high performance, with no guarantees on worst-case timing (latency). With multiple actors allowed simultaneous access to the main memory, the use of bandwidth is subject to heavy contention. System interconnection takes care of guaranteeing coarse-grained *quality of service* levels (fairness, priorities) even in presence of high contention. Most platforms place rigid limits on the maximum bandwidth usable by each coarse-grained actor, i.e., each CPU or GPU cluster, even when the other actors are not fully utilizing their allotted bandwidth. Despite such mechanisms, previous work has largely shown that heavy interference on the main memory in modern heterogeneous SoCs results in uncontrolled latency inflation [31] [10].

Table 1 presents some of the values for the bandwidth requests of the different engines on commercial SoCs. The rightmost column indicates the nominal main memory bandwidth as reported in the official datasheets. The column to its left shows the maximum bandwidth request (measured) that all the CPU cores from all CPU clusters can generate. Further moving to the left there is the bandwidth request generated by a single core from a given cluster.

Comparing the two rightmost columns indicates that a significant share of the total SoC bandwidth is reserved to other devices than the CPU (i.e., the GPU or other acceleration logic). Focusing on the CPUs (third- and second-to-last columns) it is evident that **a single CPU core only consumes a fraction of the bandwidth budget allotted to the CPU complex**.

The available bandwidth for each CPU cluster is usually sized so as to satisfy the typical bandwidth demand of the cluster. Yet, in virtually no cluster a single core may generate such a maximum demand alone. As we will show in

the experimental results section, a single ARM A57 core in the NVIDIA Jetson TX2 uses up to 66% of the bandwidth available to the whole cluster, and the Denver core only slightly more than 50%.

2.1 Predictable Execution and Memory Underutilisation

The relevance of this observation becomes clear when we consider the recent efforts that the research community has put into making the adoption of modern HeSoCs feasible in the context of real-time applications [7, 16, 37]. The stricter the real-time requirements, the more modest the adoption of parallel systems has been so far in this domain. The reason for this has to be searched in the already discussed adverse effects of resource sharing on the timing behavior of applications. Well-established methodologies for timing analysis are severely complicated when parallel and shared resources are considered, to the point of making system certification unfeasible.

A number of approaches that tackle the problem build upon the notion of removing by design the interference, thus allowing the use of static timing analysis of parallel programs. Certification authorities from the avionics domain have published a position paper for multi-core processors (CAST-32A) providing guidance for software planning, development and verification in multi-cores [2]. In these approaches it is guaranteed that the parallel threads of execution take a disciplined approach in accessing the shared system resources that impact timing under high contention. Specifically, the threads are guaranteed to access such resources in a mutually exclusive manner by design. One notable practical example of such approach is the Predictable Execution Model (PREM) [24] [5]. PREM avoids interference by design by partitioning programs into contention-sensitive *memory phases* and contention-free *computation phases*, and scheduling these such that two *memory phases* are never executed in parallel. By scheduling only a single memory phase at a time, contention at the main memory level is effectively avoided. This allows a system designer to tightly bound memory access latencies, leading to shorter worst-case execution times (WCET) and, ultimately, less pessimism in traditional timing and schedulability analysis.

Getting back to the observation that a single core in a modern HeSoC only uses a small fraction of the available bandwidth for the CPU cluster it belongs to, the *one-core-at-a-time* memory arbitration model implied by PREM-like approaches is bound to severely under-utilize memory bandwidth. Previous work has studied the admission of more than one task at a time to DRAM in a PREM system [35], but the proposed approach is too coarse-grained and does not show enough flexibility to be successful in all cases. As we will discuss throughout this paper, for such techniques to be successful the rate at which the new task injects its own transactions should be controlled in a fine-grained manner.

3 SUPPORTING CONTROLLED MEMORY REQUEST INJECTION IN HESOCS

Controlled Memory-Request Injection (CMRI) [11] has been recently proposed as a conceptual model to improve DRAM bandwidth utilization for PREM workloads deployed on multi-core CPU. The problem of memory bandwidth under-utilization is even more stringent in a modern HeSoC, where the DRAM bandwidth is designed to sustain requests from the accelerators and is is scarcely used by the CPU cores. Here we discuss two practical schemes to support CMRI on a HeSoC. Although its potential benefits extend to a broader set of usage scenarios, CMRI has been proposed as a complementary technique to a timing guarantee-preserving execution model like PREM.

A PREM implementation consists of two main components: (i) compiler support, to transform the code with undisciplined memory access in a sequence of intervals made of memory and compute phases; (ii) runtime system support to intercept the calls from the PREM-ized code to notify the beginning/end of memory and compute phases and provide proper synchronization to guarantee mutually exclusive access to the DRAM. Optimal scheduling blocks can

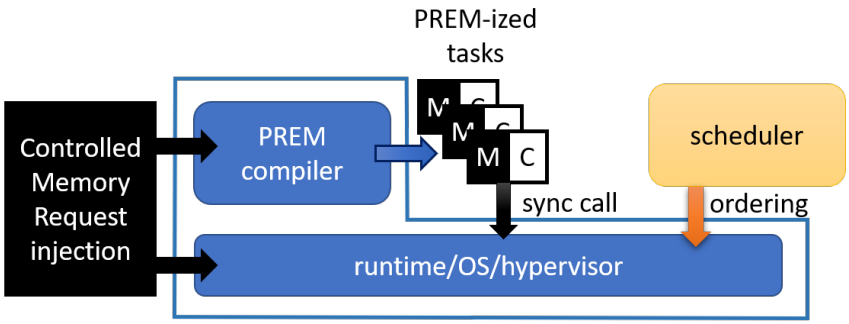

Fig. 2. PREM components and CMRI.

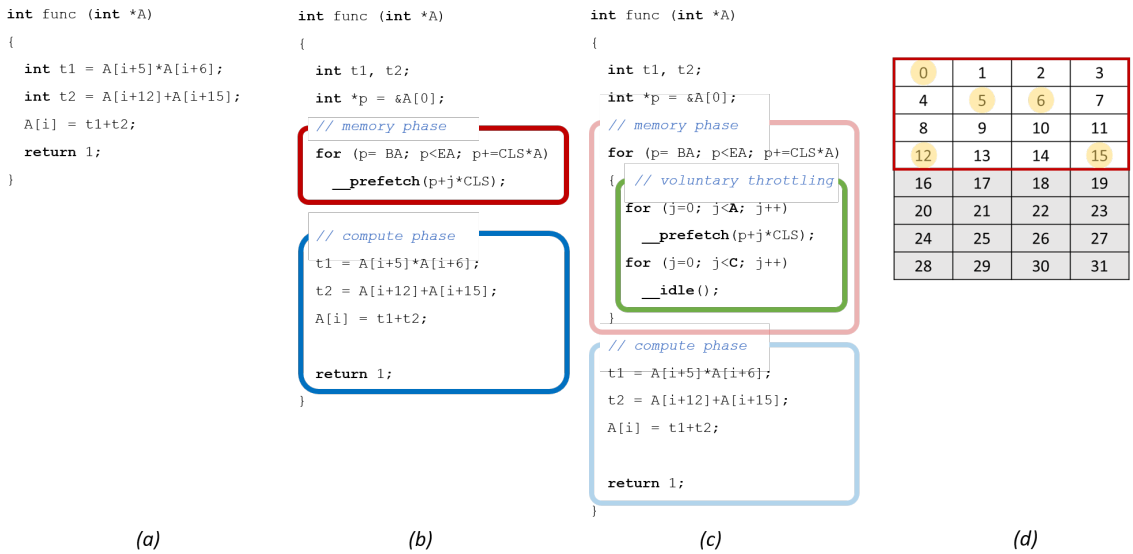

Fig. 3. An example of access pattern generation for *Voluntary Throttling*-based CMRI. (a) A code sample. (b) PREM-transformed code, with *memory* and *compute* phase. (c) *Voluntary Throttling* on top of PREM memory prefetching. Access pattern for the code sample.

be added on top of this basic support to maximize system performance. Controlled Memory Request Injection (CMRI) can also be built on top of basic PREM support, and in particular it can be implemented as an extension to the compiler or the runtime component. We describe in the following the two approaches.

3.1 Compiler-based Voluntary Throttling

PREM relies on source code transformations to implement the concept of mutually exclusive DRAM access. The key steps a PREM compiler undertakes are the following [16]:

(1) **Partitioning:** the program is divided into smaller units from which PREM intervals can be built. This typically relies on some extended representation of the *control flow graph* (CFG), like the *single-entry, single-exit* regions;

(2) **Footprint Calculation:** each of the units is annotated with the size of its memory footprint, i.e., the amount of memory the associated code uses;
(3) **Interval selection:** units are grouped into larger intervals according to some maximization heuristic – typically the largest cumulative footprint that fits in the cache, as this minimizes synchronization overheads;
(4) **Phase Generation:** the code associated to the selected intervals is transformed into compute and memory phases;
(5) **Runtime Hooks Generation:** scheduling hooks are inserted in the form of calls into the runtime system.

For performance, the *Footprint Analysis* takes into account the cache line size, and generates prefetch patterns that mimic the bursty copy behavior of a DMA. Figure 3.d illustrates this concept for the code snippet in Figure 3.a, considering a cache line size of four words. In this example, five memory locations are accessed, with a sparse pattern over time. The footprint analysis does not consider single-word accesses, but rather reasons at the cache line granularity and infers the shape of the smallest region that encloses all the cache lines (the red rectangle in Figure 3.d). From there, the *memory region* is created by generating *prefetch* commands from within a loop that traverses this region (i.e., addresses that are spaced exactly one cache line). This can be seen in Figure 3.b. Note that for a cache-based PREM implementation the *compute phase* basically reuses the original code without further transformations. The original load/store instructions are guaranteed to hit in the cache thanks to the *prefetch* commands.

The creation of a *memory phase* being under control, CMRI can be simply supported by interspersing *prefetch* instructions at regular intervals with calls to a routine that leaves the task code to idle. We call this scheme *Voluntary Throttling*. Let A be the statically-assigned length of the access interval (i.e., the number of *prefetch* instructions), and C be the length of the idle cycle (i.e., the number of NOP instructions inserted in the code). *Voluntary Throttling* thus controls injection by means of the *load intensity*, which is defined as $L = A/(A + C)$. The code transformed according to this policy is shown in Figure 3.c. Note that C is dynamically configurable by the PREM runtime, thus allowing full speed ($C = 0$) when the task is the main PREM task, and lower-paced profiles when acting as a self-limited task, called *injector* in [11].

Note that even if PREM compilers try to generate sequential patterns for their memory phases, like the example described here, irregular patterns are unavoidable in certain applications (for example when indirection is used and pointer aliasing fails to determine the exact destination of the load/store commands). In those cases the compiler conservatively includes more *prefetch* commands than necessary, and it fails at enforcing an ordering that ensures sequential behavior in the memory access pattern. In our experiments, we model this worst case situation by considering a randomic access pattern.

3.2 Hypervisor-Controlled Bandwidth Regulation

CMRI can also be supported in a transparent manner with a runtime component. This solution is mandatory in those cases where the source code of an application is not available, and thus the compiler-based approach of VOLT is not feasible. In virtualized environments it may be desirable to have injectors be unaware of the PREM execution applied in other cores. To realize this type of solution a bandwidth regulation scheme can be supported in the underlying system software.

MemGuard [37] is a runtime approach that guarantees bandwidth allocation to different CPU cores, thanks to a memory bandwidth regulator. This component is implemented with performance monitoring units that are commonly

available on modern processors and SoCs, and that enable the configuration of active notifications triggered by configurable events.

The bandwidth regulation mechanism underlying CMRI can be built on top of a virtualized environment, where an hypervisor module can manage the memory accesses of each Virtual Machine (VM). Implementing this in an hypervisor is more beneficial than at operating system level, since the first is usually running on a different core, and the overhead of its management routine is mostly negligible. Each VM i is associated to a configuration, i.e. a pair $\langle T_i, B_i \rangle$, where B_i is a *budget*, i.e., a (maximum) number of DRAM accesses, and T_i the budget refill *period*, a time duration in µs. Together they represent the maximum bandwidth allowance for i: no more than B_i every T_i. From the technical viewpoint, the bandwidth regulation is operated by two tasks.

- *Throttling routine.* A sporadic task activated on any cores assigned to i whenever $B_i = 0$. This task puts VM i in an endless idle loop, preventing it from accessing memory until the next refill event.
- *Refill service routine.* A periodic task with period T_i which refreshes the memory request allowance for i, called *budget* and denoted as B_i. This task possibly resumes regular execution, concluding the throttling routine and letting the VM continue with the execution and memory accesses.

Both tasks are executed on the core of interest, in hypervisor space, i.e. at Arm Exception Level 2, and are activated by IRQ interrupt—a timer for the refill, a Performance Memory Unit (PMU) event for the throttling.

4 AN ASSESSMENT OF CMRI COSTS AND BENEFITS

To assess the costs and benefits of CMRI in the context of HeSoCs, we consider two distinct setups. First, we conduct an exhaustive characterization based on synthetic workloads designed to stress the worst-case memory access pattern, and more specifically carrying out:

A) An evaluation of the overall effect of CMRI both *within* a single multi-core CPU cluster and *between* different CPU and accelerator clusters.
B) A comparison of *Voluntary Throttling* (VOLT) and *Bandwidth Regulation* (BWR) in terms of degree of control of the injection/overheads.

Second, we consider a large set of real benchmarks from the Polybench suite [28] and study the effect that CMRI has on the latency of such workloads. Again, we consider the effect of CMRI between multiple CPU and accelerator clusters.

4.1 Synthetic Workloads

Hardware – To assess the benefits of CMRI we choose the NVIDIA Jetson TX2 as a reference HeSoC. The block diagram of the architecture is shown in Figure 4. The *host* processor is organized in two *compute clusters*: a quad-core ARMv8 Cortex-A57, and a dual-core ARMv8-compliant *DENVER* processor (NVIDIA proprietary design). In the following, we refer to the former as the *ARM cluster*, and to the latter as the *DENVER cluster*. Each of the A57 cores features a 32 KiB L1 data cache and a 48 KiB L1 instruction cache, while each DENVER cores respectively has 64 KiB and 128 KiB. The cores in each cluster locally share a 2 MiB L2 cache.

The main system memory is an 8 GiB LPDDR4 128 bit DRAM with a total bandwidth of 59.7 GiB/s, needed to sustain requests coming from the two *clusters* and the GPU.

System Software – The Tegra X2 (TX2) is flashed with Nvidia Jetpack 4.3 featuring Linux for Tegra version 4.9.140.

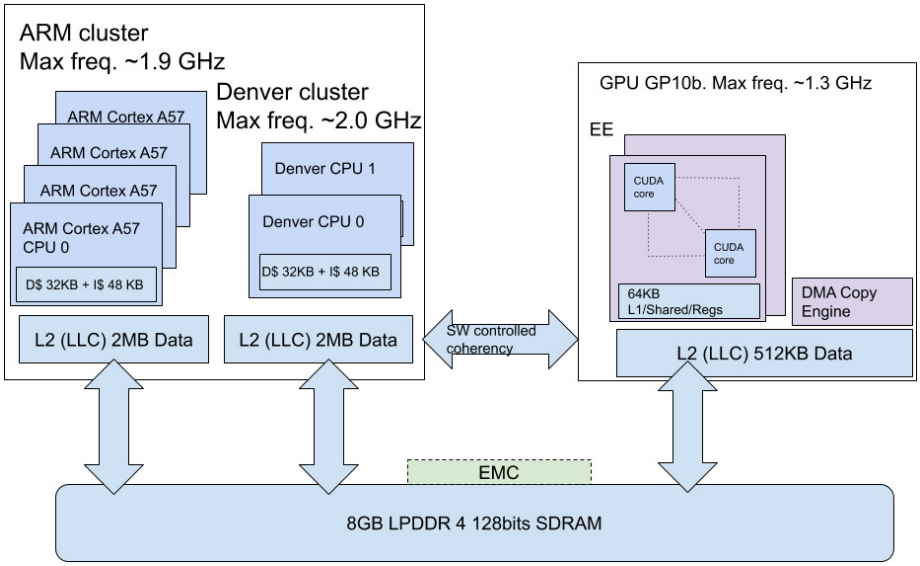

Fig. 4. Block diagram of the Tegra X2 architecture.

The System on a Chip (SoC) power management and frequency regulators, acting on CPU and memory controller, are set for maximum performance with the `nvpmodel` and `jetsonclocks` tool.

Jailhouse[17][26] was chosen as a free and open-source (GPL licensed) bare-metal hypervisor, as its design is focused on safety-critical and real-time environments. Jailhouse is built around the principle of static partitioning of hardware resources (i.e., it does not provide any support to resource overprovisioning). The main benefit is that the hypervisor is essentially absent after the hardware initialization: VMs are allowed direct, unmediated and exclusive access to the assigned hardware resources. We configured Jailhouse to host a benchmark bare-metal application on each core. Jailhouse features unofficial support to several real-time-oriented experimental features for memory management, arbitration and isolation. Amongst these, an implementation of MemGuard-core, which we adapt for the implementation of our bandwidth regulator.

Experiments – In the following we describe two types of experiments:

A) An evaluation of the overall effect of CMRI at the intra-cluster (both ARM and DENVER clusters) and inter-cluster levels.
B) A comparison of *Voluntary Throttling* (VOLT) and *Bandwidth Regulation* (BWR) in terms of degree of control of the injection/overheads;

Concerning the second point, as we have explained in Section 3.2 an implementation of MemGuard can rely on *refill* and *throttling* routines that are executed in hypervisor space (i.e., in Arm Execution Level 2) and triggered by a hardware event – a timer for the refill, an IRQ raised from the Arm Performance Memory Unit (PMU) for the throttling. The NVIDIA proprietary DENVER cores on the TX2 have a custom PMU that does not fully conform to ARMv8. In particular, the cache miss counter used for MemGuard is not implemented, hence MemGuard cannot be readily implemented on

the DENVER cores. For this reason, the first experiment only relies on VOLT as a CMRI technique. The comparison between VOLT and BWR is conducted within the ARM cluster.

Benchmarking Methodology – The focus of our experiments is on assessing the benefits of using CMRI on top of PREM workloads, under the most pessimistic conditions.

The main metrics of interest are: (i) the latency of the task currently allowed by PREM to access memory, i.e., the task *under test* (or, the lengthening of its *memory phase* while other tasks generate controlled *interference*); (ii) the bandwidth utilization reached during the execution of these concurrent memory phases.

Note that both metrics are unaffected by whether *compute phases* are executed in parallel too, or not. These metrics are also independent of the exact synchronization and scheduling mechanisms used to enforce mutually-exclusive DRAM access. We analyze in depth what happens during a single *memory phase*, for all combinations of *sequential* and *random* memory access patterns. These are representative corner cases for what could happen on any real workload. As such, we explore both patterns on both the task under test and the other tasks injecting controlled amounts of memory requests.

The NVDIA TX2, like most modern HeSoCs, features a number of hardware blocks aimed at improving the average-case memory transaction performance (cache prefetchers, DRAM row buffers, etc.). To control the bypassing of such mechanisms, our micro-benchmark implements a simple pointer walk over a portion of a statically pre-allocated, large array (64MiB) of data structures, each containing: (i) a pointer to the next address to read/write; (ii) padding, to fill the remainder of a whole L2 cache line. This is needed to effectively model *sequential* and *random* memory access patterns.

For each configuration of interfering-task workloads, we repeat our experiment twice: (i) once for measuring the bandwidth requested by the tasks, (ii) once for measuring the memory access latency experienced by the task under test. To obtain a sensible bandwidth measurement, it is necessary to observe a relatively long time; on the other hand, latency measurements are actually measuring individual PREM memory phases, which are relatively short.

We built a micro-benchmark, which we called *mem_bench*, able to deal with each of the above two types of measurements. This benchmark is an extension of the `lat_mem_rd` test from the LMBench suite [21][1]. *Mem_bench* can be configured to operate in two modes:

(1) ***mem_bench_LAT*:** we use this mode to measure the latency of the PREM task *under test*. In this case, the benchmark reads or writes the data specified as the PREM task memory phase only *once*. We run each experiment 100 times, and we take the worst-case value (95th percentile to filter out outliers). This is the most accurate mode for latency measurements;
(2) ***mem_bench_BW*:** we use this mode to measure the average bandwidth request generated by the *interference* and PREM *under test* tasks. In this case, the benchmark continuously reads or writes the data *multiple times* to ensure the duration of the transfer is long enough (60 seconds) to allow for a stable and meaningful measurement. This is the most accurate mode for bandwidth measurements.

To model the worst-case interference that the *interference* tasks can generate, we execute them for the whole time that the task *under test* is running[2]. In a real system the interference suffered by the task *under test* would in general be smaller, and the benefits of CMRI higher.

[1]Available for download: https://git.hipert.unimore.it/msolieri/membench.
[2]Specifically, *interference* tasks start execution before we launch the task *under test*, and complete execution after the task *under test* has terminated.

4.1.1 Experiment A: Assessing the Benefits of CMRI. Considering the cluster-based nature of the target hardware, we conduct our experiments in three settings:

(1) inside the ARM cluster;
(2) inside the DENVER cluster;
(3) across the two compute clusters.

For experiments 1 and 2, we consider a single task *under test* (UT), which represents the task that a regular PREM scheme would grant exclusive access to memory. On top of that, we explore the effect of allowing CMRI from a number of *interference* tasks (IF), mapped on the remaining cores from the same compute cluster (each task is pinned to a different core). For these, we vary (with exponential spacing) the LOAD intensity from near-zero to 100%.

The size of the ARM and DENVER cores' *memory phases* is set to 512KiB and 1024KiB, respectively (the size of the L2 cache divided by the number of cores in each cluster, which is a fine grained choice for the memory phase size, not manageable by tools such *memguard*)

The third set of experiments studies the effect of interference among the two compute clusters. One ARM core and one DENVER core run one of two UT tasks. We run four IF tasks on the remaining cores (three Cortex-A57, one DENVER) from both clusters. The GPU is active and generates as many *sequential* memory requests as possible, since we want to experience the worst possible situation for the UT task, with all the available engines requesting the maximum bandwidth at the same moment.

To stress all the possible corner cases, we configure *mem_bench* to generate eight different combinations for UT and IF tasks, considering both read-only and read+write workloads, and both *sequential* (SEQ) and *random* (RAN) access patterns. The workloads of the IF tasks are implemented by means of mem_bench_BW, while the UT task is run two times with different configurations for each experiment: mem_bench_LAT when measuring latency and mem_bench_BW for bandwidth. From our experiments, we observed that independent of the fact that we used read-only or read+write workloads, the range of applicability and magnitude of effects of CMRI is essentially the same. Consequently, due to the lack of space (and for the sake of presentation) in the following we only present and discuss results for reads.

In details, we consider the following four combinations:

(1) UT=SEQ, IF=SEQ;
(2) UT=SEQ, IF=RAN;
(3) UT=RAN, IF=SEQ;
(4) UT=RAN, IF=RAN.

For each of these combinations, our plots show a breakdown of the bandwidth usage among different cores (stacked areas, to be read on the left Y-axis) and the latency increase experienced by the UT task (lines with markers, to be read on the right Y-axis) as we increase the amount of injected IF traffic by controlling the load intensity through the value of A and C. Specifically, we select $A = 1$ (a single cache line is read) to allow for the finest granularity, whereas C (the number of NOPs between one cache line read/write and the other) is varied from 0 to 4500 in discrete steps at an exponentially increasing distance for a total of around 100 points.

As a main metric to measure CMRI we introduce the concept of *injection rate* (shown in the top X-axis of every plot), defined as the ratio between the bandwidth requested by an IF task for a given *load intensity* and the bandwidth obtainable at full throttle (i.e., *load intensity*=100%) by the same task. We sweep *injection rate* by 10% intervals (20% to

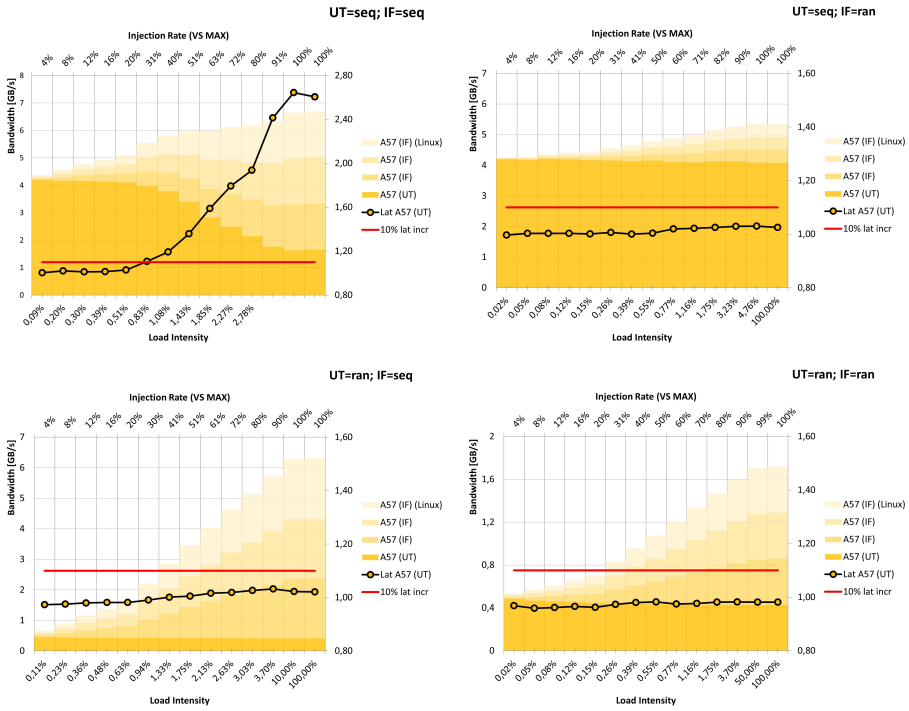

Fig. 5. Effects of CMRI within the ARM cluster.

100%)[3]. Below 20% we increase the sampling resolution (at 4%, 8%, 12% and 16%) to observe the effects in an area where the granularity of the technique becomes really fine (and thus we expect to have less control/more overhead). The *load intensity* corresponding to each *injection rate* value is also reported in the plots for reference (bottom X-axis). Note that we report two values at 100% *injection rate*: the first is the one achieved with minimum *load intensity*, the second with 100% *load intensity*. The closer these two values, the less margin CMRI has to saturate the bandwidth.

The plots also show a red line which represents an upper bound to the tolerated increase in the UT task latency – which we set to 10%[4]. This allows to easily spot how much CMRI can be used in each configuration before this point is reached.

Intra-cluster: ARM Cluster Figure 5 shows the plots for CMRI usage inside the ARM cluster under all the considered combinations of traffic types. Focusing on bandwidth usage (stacked areas), in all the plots it's evident that a significant portion of the available bandwidth – which is not exploited by the PREM task alone (as shown when the *injection rate* is at its minimum) – can be utilized by CMRI (as shown as we move to the right). By combining the information from the bandwidth areas and the latency curve it is possible to identify the sweet spot where, for each traffic pattern combination, we can gain the most.

When both the task *under test* and the *interference* traffic feature sequential access pattern – which is the most sensitive case to interference – the cumulative bandwidth request from all the ARM cores (100% *injection rate*) is

[3]The actual *injection rate* values in the plots sometimes slightly differ from the nominal values at 10% intervals, as the specific (A,C) values used for the experiment might not exactly produce the desired *injection rate* values. We report in the plots the closest ones to the nominal.
[4]This value clearly depends on the specific application requirements and is only provided as a reference.

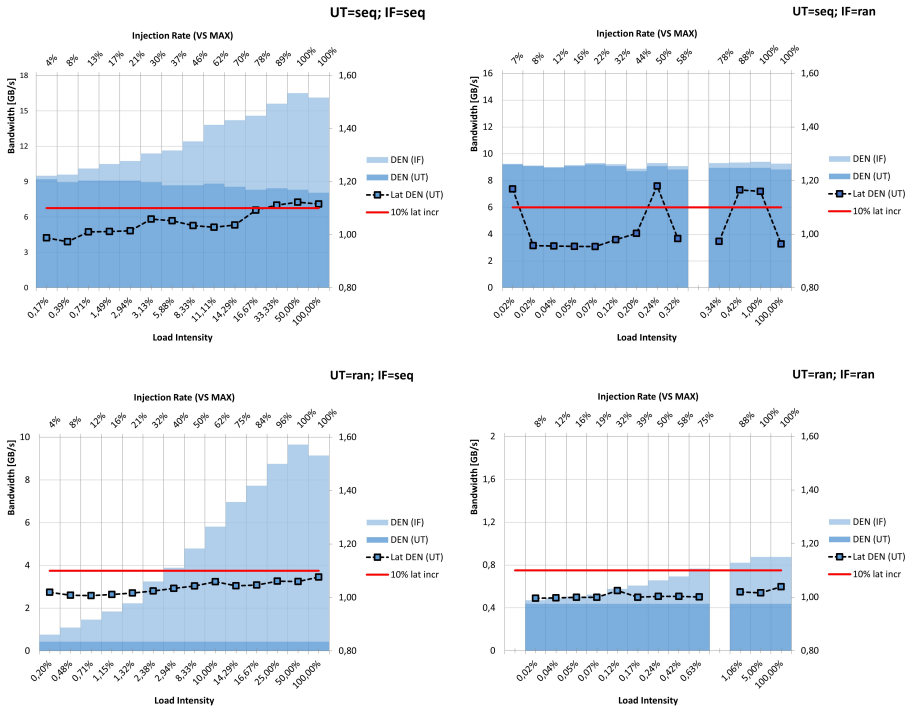

Fig. 6. Effects of CMRI within the DENVER cluster.

limited to only around 6.6GB/s. The plot reveals that this is due to system-level bandwidth *capping*, as the latency increase at this point is at 2.5× (a single core in isolation utilizes 66% of that maximum bandwidth, as shown for 0% injection). Increasing CMRI up to 30% keeps the latency increase within the 10% threshold. Note that 30% *injection rate*, corresponding to ≈0.8% *LOAD intensity*, allows to reach 83% of the maximum bandwidth allowed within the ARM cluster (6.6GB/s) with little impact on the execution time of the UT task.

When the IF workload is of type RAN, the SEQ UT task is never perturbed, even with 100% CMRI. This brings a 26% increase in the overall ARM cluster bandwidth usage, equivalent to 80% of the maximum. When the UT task features RAN traffic it is mostly insensitive to the IF tasks activity. When IF tasks employ SEQ traffic CMRI enables a tremendous 9.4× improvement on the cluster memory bandwidth usage, reaching 94% of the maximum. Also when the IF tasks employ RAN traffic the bandwidth requests can be fully summed up, increasing bandwidth usage by 3.22× without impacting the execution time of the UT task.

Intra-cluster: DENVER Cluster Figure 6 shows the results for CMRI within the DENVER cluster. The benefits of CMRI here are even more pronounced, as one single DENVER core in isolation uses only slightly more than half the available bandwidth for the cluster (as opposed to 66% for a single ARM in the quad-core ARM cluster). Focusing on the worst-case SEQ-SEQ experiment, it can be seen that the increase in the latency of the UT task stays below 10% for CMRI as high as 80%. The bandwidth usage reaches nearly full efficiency, as it gets nearly doubled.

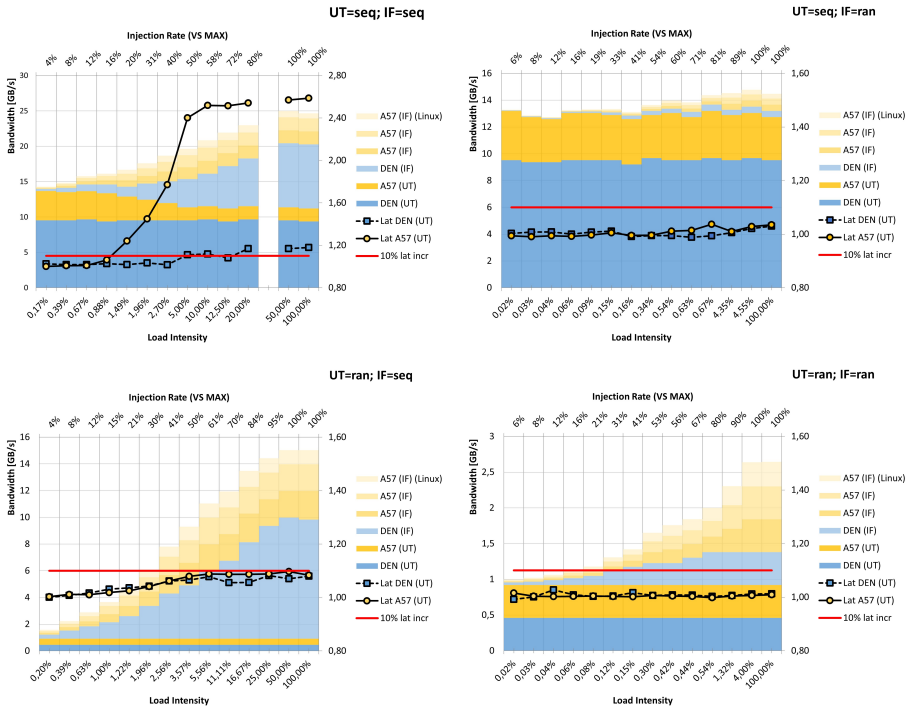

Fig. 7. Effects of CMRI across the ARM and DENVER clusters.

Similar to the ARM cluster, there is not a lot to be gained when the IF task is of type RAN and the UT task is of type seq, as the bandwidth generated by the random access pattern is an order of magnitude smaller than the sequential for the DENVER core. Intuitively, the latency increase should stay well below 10% and nearly unmodified in this case, even for 100% CMRI, as shown by the modest increase in bandwidth. However, it must be noted that the latency measurements for this experiment (and only for this experiment) suffered from very high variance[5], which is probably to be attributed to a peculiar yet systematic effect of noisy setup.

Similar results to the ARM cluster apply when the UT task is of type RAN, where CMRI enables huge improvements in intra-cluster bandwidth usage. If the IF task is of type SEQ, we increase bandwidth usage by 12×, and if the IF task is of type RAN, we can double the bandwidth usage, in both cases without impacting the UT task at all.

Inter-cluster: SoC-level Effect of CMRI After studying the benefits of CMRI within each compute cluster in isolation, we experiment with inter-cluster interference. We first elect a single ARM or DENVER core, in turn, to host the UT task, and we place the IF tasks on the sole cores belonging to the other cluster. Thus, when one DENVER hosts the UT task, the IF tasks run on the ARM cluster (the second DENVER core is inactive), and vice-versa. We don't show plots for this experiment (which is preliminary to the following insights), but we discuss here the most important findings.

[5]This is not just an increase in the number of outliers, as considering only the 50th percentile does not reduce significantly the variance.

Table 2. Test configuration summary

	CMRI technique				
	VOLT (A)		*BWR (T)*		
Test app.	1	16	2 us	8 us	32 us
mem_bench_BW	✓	✓	✓	✓	✓
mem_bench_LAT 512 KiB	✓		✓		✓
mem_bench_LAT 32 KiB	✓		✓	✓	
mem_bench_LAT 8 KiB	✓	✓	✓		

Other params.	*Possible values*
UT task pattern	sequential, random
IF task pattern	sequential

When a DENVER core hosts the UT task, its latency is virtually unmodified (<5%) independent of the *injection rate* of the IF tasks running on the ARM cores. When it is an ARM core that hosts the UT task, its latency is more susceptible to the activity of the IF tasks (running on the DENVER cores), but the variation always stays below 10% if the *LOAD intensity* stays within 33%.

These findings suggest that the best way to support PREM on a platform of this type is that of always allowing one core from each cluster to access memory. This still leaves plenty of room for better bandwidth exploitation, which CMRI can achieve.

Figure 7 shows the results for an experiment where an ARM core and a DENVER core both run an UT task, while the remaining cores from both clusters run IF tasks. At the system level, the benefits of CMRI already observed within each cluster are confirmed. When both the UT and IF tasks run SEQ traffic we can tune IF tasks to inject with a rate of up to 16% to avoid perturbing the ARM UT task beyond the 10% latency increase threshold. This still brings a 13% increase in bandwidth usage compared to only allowing a single Cortex-A57 and a single DENVER core to access memory (a basic PREM scheme). The DENVER UT task could tolerate up to 40% *injection rate*, bringing the bandwidth usage improvement to 31%. If the IF tasks are of type RAN, no significant interference is generated on the UT tasks on both clusters. The benefits are more modest (as already observed within the individual clusters), with an increase in bandwidth usage of around 11%. Note that the values in the latency curves (in particular for the DENVER) might sometimes drop below one. This is due to the fact that, as a baseline value for normalization, we consider the latency measured when a single core from a given cluster executes (SEQ or RAN) while all cores from the other cluster (and from the GPU) execute SEQ IF tasks, as explained at the beginning of this section. In this particular experiment, we have less interference coming from the other cluster, as one of the cores hosts the UT task instead of an IF one. As a consequence, particularly when the UT task is of type RAN, the amount of interference generated for the other cluster is smaller.

When the UT tasks are of type RAN, we see again the highest potential for making a better use of the available bandwidth. Independent of the fact that the IF tasks are of type SEQ or RAN, they can inject at full throttle, achieving an improvement in system-level bandwidth usage of 9.5× and 2.65×, respectively.

4.1.2 Experiment B: A Comparison of VOLT and BWR Implementations. The second experiment set is aimed at comparing the two CMRI techniques that we have described, to assess their effectiveness at controlling the *injection rate* and their sensitivity to implementation overheads. For this reason, different from the setup for the previous experiment,

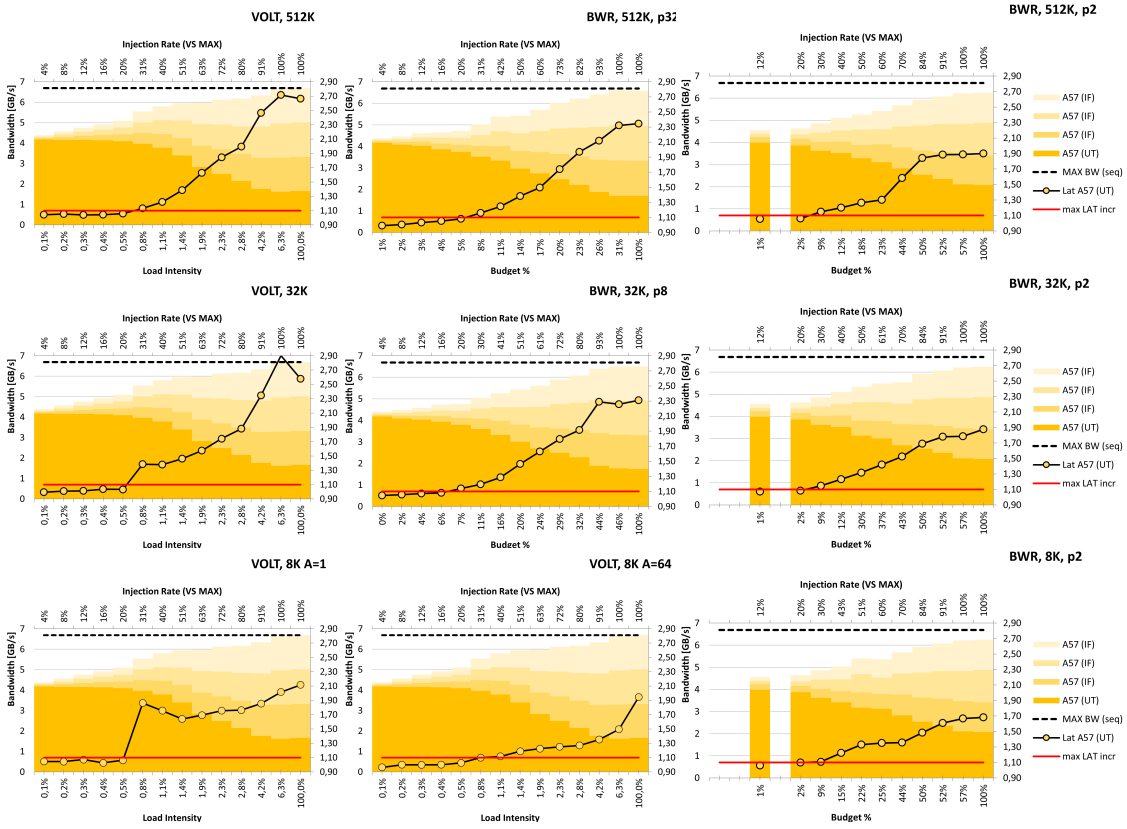

Fig. 8. Comparison of VOLT and BWR for varying *memory phase* sizes, A and T. UT and IF task traffic is of type SEQ.

we consider different values for numerous system parameters. The sizes for the *memory phases* of PREM tasks are considered in three variants, to cover several possible use cases, and in particular:

(1) largest at 512 KiB: This equals ¼ of the L2 cache size on the ARM Cluster, and represents the same maximum-size heuristic used in the previous experiment;
(2) medium at 32 KiB: This equals the size of the L1 cache for ARM cores. It is representative of a PREM scheme implemented at the L1 rather than at the L2;
(3) smallest at 8 KiB: This represents the smallest reasonable size below which the measured overheads to implement memory arbitration would dominate (see timing measurements later).

Concerning VOLT, similar to what we have already done in the previous experiment, we vary C to produce a varying *load intensity* as a means of controlling the *injection rate*. In addition, in this experiment we also consider different values for A, as we will explain later on.

Concerning BWR, for any bandwidth limitation target there exist several $\langle budget, period \rangle = \langle B, T \rangle$ configuration pairs to express it: a correlated increase of *budget* and *period* leaves the bandwidth unchanged. Intuitively, the shorter the *period*, the finer-grained the control, since it is most frequently updated. On the other hand, shortening the *period* makes

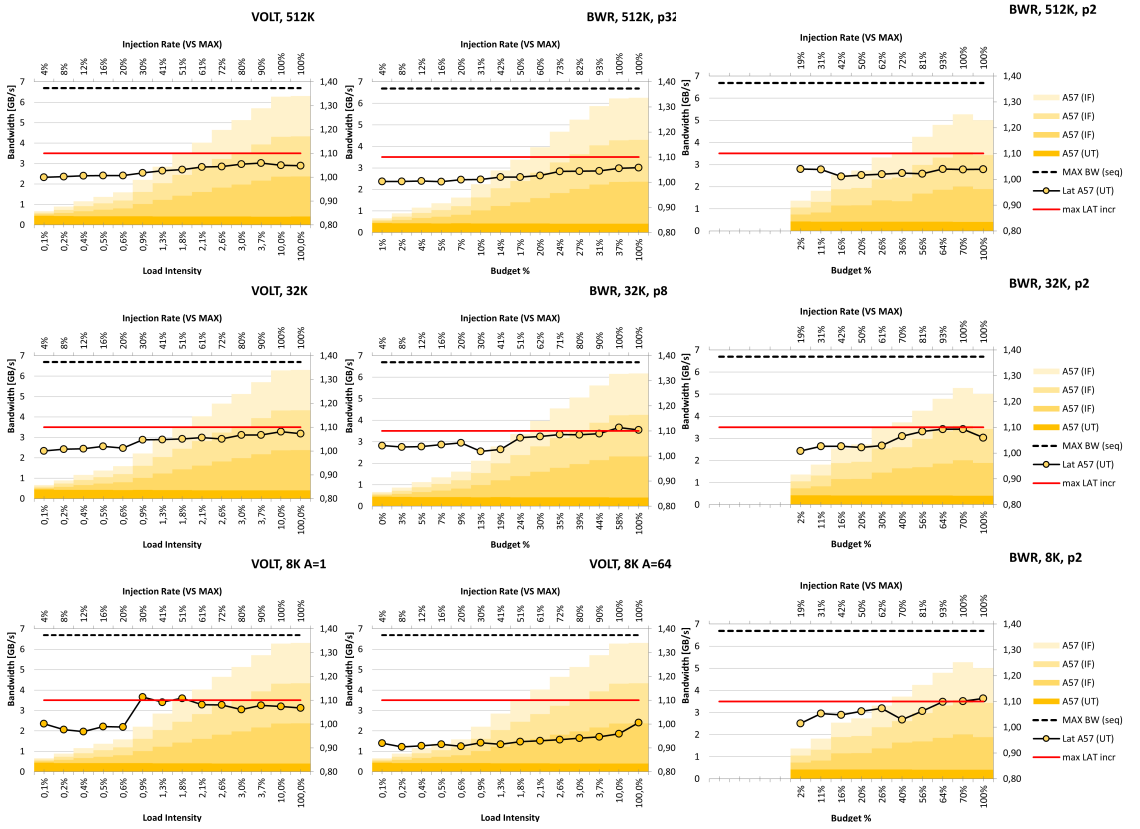

Fig. 9. Comparison of VOLT and BWR for varying *memory phase* sizes, A and T. UT task traffic is of type RAN, IF task traffic is of type SEQ.

the technique more sensitive to overheads, because the execution latency of the refill routine, say L_{refill}, increasingly eats the available time for memory usage, i.e. $T - L_{\text{refill}}$.

The *budget* may also become meaninglessly large for the given *period*. For any T there exists an upper-bound budget B^T such that for any $B \geq B^T$ the bandwidth limitation obtained with $\langle T, B \rangle$ is approximately equal to $\langle P, B^T \rangle$. This upper bound budget is essentially the maximum amount of memory requests that the platform is able to serve within a time $T - L_{\text{refill}}$. The *budget rate* is thus defined as B/B^T.

To tune MemGuard as an injection mechanism we benchmarked some key figures of the Jailhouse implementation on the ARM cores. We started from profiling L_{refill} over 10 K test runs. The test routine has been implemented as a bare-metal application which measures the following steps: (i) interruption by the MemGuard server in EL2; (ii) budget refill; (iii) return to EL1 execution. We obtained 0.593 µs in the average case, and 1.75 µs in the worst one.

This allows us to choose three testing values for T.

- *Short.* $T = 2$ µs: the minimum value capable of accommodating at least the refill routine.
- *Medium.* $T = 8$ µs: approximately 0.5 µs below the isolated latency of the medium PREM phase (32 KiB).
- *Long.* $T = 32$ µs: close to 0.25× the isolated latency of the large PREM phase (512 KiB).

For each T' of the values for T, we found $B^{T'}$ and chose n values for $B \in (1, B^{T'})$ such that $n \gtrsim 100$ and the distribution within such interval is exponentially decreasing with the budget.

Table 2 summarizes the configurations considered for this experiment. Due to space limits, and to simplify the discussion, we only show results for IF traffic of type SEQ, where CMRI is most effective, as we have learned in the previous section. VMs running IF tasks are started first by Linux 4 Tegra (L4T), which is allocated on a Denver core, then the VM hosting the UT task is spawned. This guarantees the cleanest setup for the measurement. To minimize the noise L4T is in a headless configuration, with only Ethernet and UART available as I/O channels. Tasks output log is emitted only after the end of the test. L4T also handles the submission of a GPU task that performs continuous SEQ requests on the main memory as previously described.

Discussion Figures 8 and 9 show the comparison plots between VOLT and BWR. The first refers to the configuration where UT=SEQ and IF=SEQ traffic, the second has UT=RAN, IF=SEQ. The plots are organized in a grid of 3×3: the rows refer to the three *memory phase* sizes, the leftmost column refers to VOLT, $A = 1$, and the remaining two columns refer to BWR with different T or to VOLT with different A.

The first thing that we can observe when we focus on a row and look at the plots from left to right is that both the cumulative bandwidth and the UT latency slightly decrease. This is an indication of the fact that BWR becomes more sensitive to the overheads as the *period* T is shortened, which confirms the initial intuition. When the UT task generates SEQ traffic (Figure 8) this is not particularly relevant, since higher injection rates than 20% produce unacceptable latency degradation. On the other hand, when the UT task generates RAN traffic (Figure 9) BWR uses CMRI slightly less effectively than VOLT if the *period* is too small. The other intuition that a shorter *period* should provide finer-grained injection control is instead completely disproved. The obtained *injection rate* is, on the contrary, coarser grained, because the shorter *period* imposes a smaller upper-bound *budget*, and this in turn limits the achievable control (as a reference, $B^{2\mu s} = 100$). This can be noted also visually by looking at the "holes" in the lower *budget %* region of the plots, which are control points that the technique could not instantiate. Note that, particularly when the UT task uses SEQ traffic, the lack of control in the lower range of the *injection rate* is indeed a limitation, as that is the only region where CMRI can be applied without significantly impacting the latency of the UT task. The lesson learned here is that T should efficiently chosen as the largest value smaller than a reasonably safe upper bound to the *memory phase* duration.

Focusing on VOLT plots (leftmost column), as the *memory phase* size gets smaller we observe a step when crossing 30% *injection rate*. This is due to a systematic latency increase of around 4 µs caused by a penalizing interplay between the particular temporal injection pattern created by VOLT with $A = 1$ and some architectural features (*burst* reads). Using higher values for A enables the bursty patterns and removes this effect, as shown in the central plot of the bottom row, where $A = 64$. Note that the higher value for A slightly alters (reduces) the obtained *load intensity* (C values are unmodified), which motivates the minor reductions in cumulative bandwidth and latency.

4.2 Real Workloads

As a final set of experiments we consider several of the Polybench benchmarks [28] as workloads under test and study the effect of applying CMRI on their latency at the inter-cluster level. A brief description of the benchmarks is provided in Table 3. To make results as general and comprehensive as possible, these experiments have been carried out on two different types of HeSoCs: one based on a GPU accelerator (the NVIDIA Tegra TX2 previously described) and one based on an FPGA accelerator (the Xilinx Ultrascale+ MPSoC). We describe in the following the setup and the results for this experiment.

Benchmark	Description
adi	Alternating Direction Implicit solver
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
conv-2d	Bidimensional Convolution
dynprog	Dynamic programming (2D)
fdtd-apml	FDTD using Anisotropic Perfectly Matched Layer
gesummv	Scalar, Vector and Matrix Multiplication
lu	LU decomposition
reg_detect	2-D Image processing
syr2k	Symmetric rank-2k operations
trisolv	Triangular solver

Table 3. PolyBench kernels used as real-world workloads.

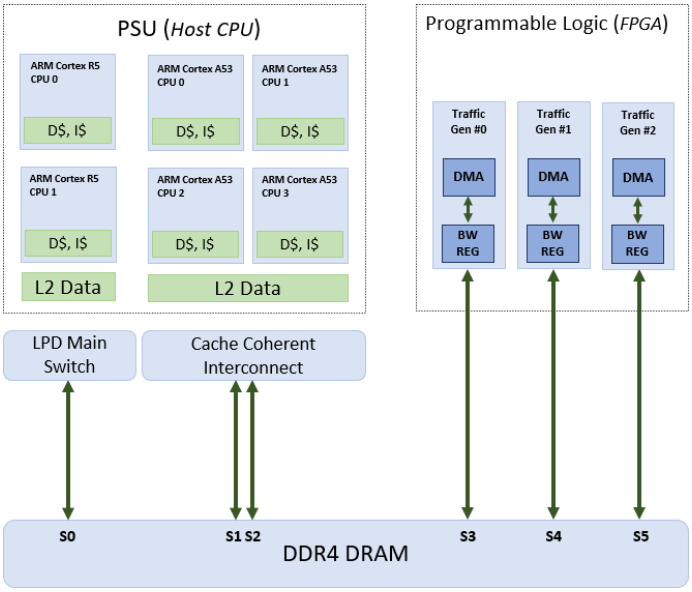

Fig. 10. Block diagram of the Zynq UltraScale+ architecture.

Hardware – For the hardware and software setup of the NVIDIA platform we refer the reader to the previous section. In addition to that platform, we also consider the ZU9EG, the flagship system-on-chip of the Xilinx Zynq UltraScale+ family. As shown in Fig. 10, the platform is composed of two different CPU clusters, namely the Real-Time Processing Unit (RPU) and the Application Processing Unit (APU). The RPU cluster, composed of two ARM Cortex R5F, has better predictability compared to the APU, but significantly lower performance, thus we do not consider it in our experiments. The APU is composed of four ARM Cortex A53 CPU cores. Each A53 core features a 32KB L1 cache for instructions and data and 1MB of L2-cache in CCI coherency domain, shared between each A53 core.

Coupled to the *host* CPU clusters an FPGA fabric can be exploited to accommodate custom logic.

The main system memory is a 4GB DDR4 64 bit DRAM with a total bandwidth of 17GB/s. As reported in the block diagram, the main memory subsystem features six DRAM ports: the first three ports (`s0` to `s2`) are reserved for the host CPUs and the remaining three (`s3` to `s5`) can be exploited by the FPGA fabric.

To correctly model the behavior of a system design that maximally exploits the DRAM bandwidth we deploy three configurable memory traffic generators, i.e., three configurable DMA engines (representative of three custom accelerators), one for each DRAM port available.

System Software – To conduct the experiments on top of the ZU9EG, we rely on a Linux environment based on the *Xilinx PetaLinux 2020.2* toolchain. The Linux distribution that we used is composed of a Linux kernel at the *5.4.0-xilinx-v2020.2* version and a `Ubuntu 20.04` root filesystem. The FPGA fabric is configured with a custom-made hardware design which couples the DMA engine to a configurable bandwidth regulator. This regulator implements the CMRI functionality for each of the traffic generators, that can be independently controlled and configured via a dedicated Linux application.

Experiments and Benchmarking Methodology – The peculiarity of HeSoCs, in terms of memory interference, is the possible presence of interference between heterogeneous clusters. For this reason, and after having studied intra-cluster interference in the previous section, in this section we focus only on inter-cluster interference. We describe in the following two experiments aimed at evaluating the CMRI technique applied to real benchmarks, taking into account the two aforementioned platforms, and for inter-cluster interference.

In detail, the two experiments measure the effect of interference between the ARM cluster and (i) the DENVER cluster plus the GPU accelerator on the NVIDIA platform; (ii) the FPGA accelerators on the Xilinx platform. The plots for these experiments show the injection rate for the interference workload on the X-axis, and the latency increase for the Polybench benchmarks on the Y-axis.

Discussion – Figures 11 and 12 show the results for the considered experiments. The various curves represent the latency increase for each of the Polybench benchmarks. For reference, we also include the curve for the synthetic workload from the previous section (SYNTH) as well as the 10% latency increase *threshold*. For the NVIDIA platform we refer to the worst case obtained for the SoC-level experiment in Section 4.1.1 (Figure 7). For the XILINX platform a similar worst-case experiment has been conducted to obtain the reference curve.

By comparing these figures we observe two main things. First, most of the Polybench benchmarks seem to suffer very little from the SoC-level interference, which suggests that these workloads have very low L2 cache miss rate. Second, the LU decomposition shows a much sharper latency increase on the Xilinx platform as the injection rate from the accelerators increases. This suggests that the workloads deployed on the GPU on the NVIDIA platform are not saturating the available bandwidth, unlike the FPGA accelerators on the Xilinx platform. In the latter CMRI is visibly required to satisfy the maximum latency increase constraints.

Comparing the behavior of the Polybench benchmarks to the worst-case curves confirms that: (i) applying PREM-like schemes that imply mutually exclusive CPU/accelerator access to DRAM heavily under-utilizes the system bandwidth (because benchmarks suffer from very little interference). In this case most of the time the accelerator could operate at full throttle without harming the CPU tasks' latency; (ii) suggests that there is a very large region where more DRAM-intensive benchmarks (i.e., those with a higher L2$ miss rate) would exceed the maximum latency increase threshold in absence of CMRI. Note that this could also happen to the same benchmarks when much larger datasets are

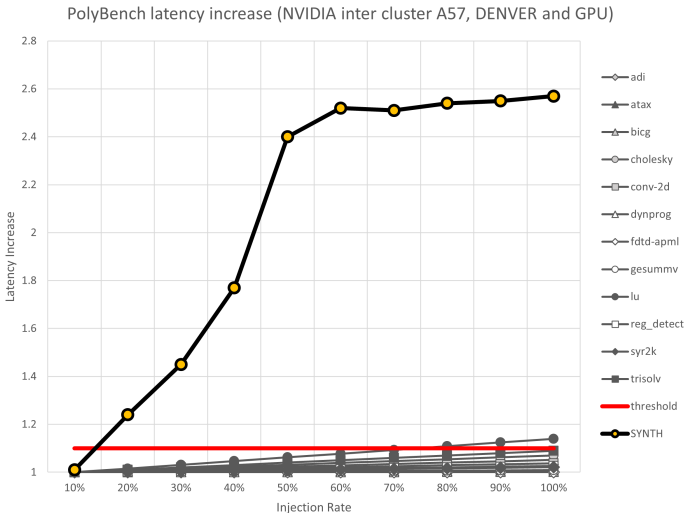

Fig. 11. Latency increase for the Polybench benchmarks running on a single ARM A57 core of the NVIDIA platform. The interference workload is synthetic and runs on both the DENVER cores (SEQUENTIAL type) and the GPU (memset).

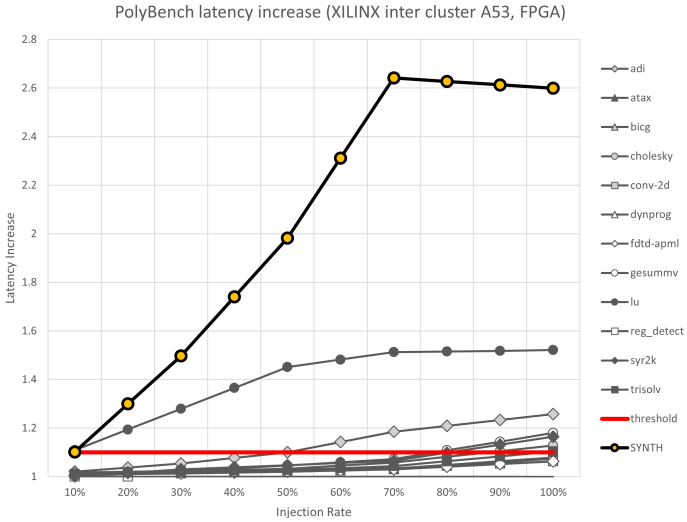

Fig. 12. Latency increase for the Polybench benchmarks running on a single ARM A53 core of the XILINX platform. The interference workload is synthetic and runs on each of the traffic generators in the FPGA (DMA copy).

used and/or when the inter-cluster interference is coupled to intra-cluster interference (which is what happens with the worst-case synthetic curves).

5 RELATED WORK

The effects of memory contention in modern system-on-chips have been abundantly discussed in previous literature. Efforts to study the deterioration in the WCET of memory-contending applications have been performed on multi-core

embedded systems [25], HPC-oriented systems [18, 29] and even the magnitude on memory interference of co-running integrated and discrete GPUs [10, 23, 32, 33] has been measured in previous work.

Memory-bandwidth partitioning schemes for guaranteeing temporal isolation have been proposed [22]. Partitioning is evidently a simple and robust solution. Yet it suffers from the bandwidth underutilization issues we highlighted in the introduction. In addition, it provides a coarser control on task execution than PREM. For example, it cannot handle constrained deadlines effectively (differently from PREM). The same issues affect also the other hardware-level partitioning solutions or bandwidth-allocation enforcement mechanisms available in the literature [1, 12, 30].

PREM has been applied to COTS multi-core systems [4, 6, 8, 34], which is also our target; after the extension of PREM to avoid inter-core interference in multi-core CPUs [5, 27], there has been extensive work on controlling CPU-GPU interference with PREM on HeSoCs [15, 16, 20].

Previous work has characterized the effects of memory interference on modern HeSoCs [10, 31], but the focus has always been on latency only, with no study of the bandwidth utilization, or the correlation between the two. More in general, none of the aforementioned contributions assess whether it is convenient on a performance and predictability perspective to arbitrate simultaneous memory accesses in modern HeSoCs through controlled injection of memory clients' requests.

In [35], the authors have improved PREM by admitting more than one task at a time to access memory, experimentally proving that the latency of main-memory accesses does grow less than linearly with the number of cores accessing memory at the same time. This PREM variation allows a statically configurable number k of Memory Phases (M-phases) to execute in parallel. Unfortunately, this is a too rigid scheme for general task sets. In fact, as we show in this paper, a certain value for k may be too high for preserving control on latency for some memory access patterns, but too low for enabling other access pattern to use a high percentage of the memory bandwidth. In the worst case, $k = 2$ may already cause unacceptable latency degradation for certain workloads, making this approach unfeasible. In other cases, the only option to preserve control on latency with heterogeneous access patterns is to set a conservative, low value of k. With such a value, bandwidth remains underutilized for some of the memory accesses.

Concerning the VOLT approach and compiler level actions, the first PREM compiler prototype was proposed in [24, 34], based on explicit annotation of intervals and all data to be transfered. Based on these annotations, the compiler generated memory phases. While this early work was able to automatically inject prefetch phases, all the heavy lifting was done by the programmer.

A non-compiler approach for PREM code generation is LightPREM [19], which replaces manual code annotation with memory access tracing, from which memory phases are constructed. In contrast to compilers, that have full visibility of the program, the LightPREM analysis needs to be executed multiple times to identify and remove all memory accesses that do not occur during every (tested) input of the program. This is key for correctness, as prefetching an invalid address would result in a segmentation fault. This means that only a subset of memory accesses of a given execution may be covered.

This article is mainly inspired by the conference paper where CMRI has been first proposed [11]. This work employs a much cleaner experimental setup (bare-metal applications on paritioning hypervisor instead of embedded Linux application). Also, it presents the first two CMRI implementations.

Yao et al. [36] proposed a memory-centric scheduler that allows different cores to access in parallel memory. They observed that the slowdown is quite low especially when the cores use different memory banks. Their analysis leverages the multiprocessor response time analysis by assuming that at most $k > 0$ cores can access main memory and the per-core memory bandwidth is constant.

Also Blin et al. [9] implement a control-mechanism for mixed-criticality applications running on COTS-multiprocessors. Courtaud et al. [14] infer the interference suffered by concurrent applications and identify interference leveles for different memory access patterns. Both works confirm our general observation that bandwidth capacity depends on the number of cores accessing memory at the same time.

6 CONCLUSION

Heterogeneous systems-on-chip couple multi-core CPU clusters and GPU (or other accelerator) clusters with shared DRAM. This is very effective for scalability and cost, and provides very high average-case performance. High contention from a number of actors on the DRAM is however subject to severe interference, which results in unpredictable latency increase. The PRedictable Execution Model (PREM), like other software approaches aimed at enabling HeSoC usage in the context of real-time systems, tackles this issue by restructuring tasks as a sequence of *memory* and *compute phases* and allowing exclusive memory access to only one *memory phase* at a time. This arbitration scheme has the downside of under-utilizing the available DRAM bandwidth in a modern HeSoC. *Controlled Memory Request Injection* (CMRI) is a technique that allows multiple tasks to access DRAM at the same time in a controlled manner, trying to use as much of the available bandwidth as possible without significantly impacting the latency of the PREM-scheduled task. In this paper we have discussed two practical schemes to support CMRI in modern HeSoCs, and presented an in-depth characterization of the costs and benefits of the technique. Our experiments on an NVIDIA Tegra TX2 HeSoC demonstrate the effectiveness of CMRI for HeSoCs, which allows up to 12× better usage of bandwidth at the intra-cluster level and up to 9.5× at the inter-cluster level. Overall, compiler-level CMRI techniques (VOLT) have smaller overhead and enable very fine-grained controlled of the amount of injection, but cannot be applied in those cases in which the source code of the application is not available. In these cases, hypervisor-level bandwidth regulation (BWR) provides very efficient CMRI support, provided that the refill period is not too small.

As future evolutions of this work we plan to explore architectural support for efficient CMRI implementation.

Although useful in static scheduling scenarios, in our view CMRI could be effectively determined dynamically during system operation. A scenario we envision, and that we are considering in our ongoing work, is one where the user can specify a desired QoS requirement for a the task under test running on a CPU – in the form of a maximum percent elongation of its typical execution time (average, or worst-case). Each task is tagged with metadata that informs the system about its operational intensity (or an equivalent memory-to-computation metric: this can come from an offline profiling step, or be inferred at runtime). Based on the characterization for the worst-case interference presented in this paper (or the equivalent for a different platform, derived with the methodology presented in this paper), an OS scheduler-level control policy can set a CMRI value for each interference task individually. This is based on the fact that the policy is aware of which tasks are ready to execute and knows the operational intensity of each task, from which it can determine the worst case maximum interference a particular co-schedule can generate for the task under test. For those cases where the timing constraints are softer, we are also exploring a solution where the CMRI is speculatively set to a default value upon the interference task launch and dynamically adjusted during its execution, based on periodic monitoring of the interference task's use of the memory bandwidth. The same mechanism can also be used to determine the task under test's operational intensity at runtime. Our preliminary results with this setting confirm that the approach is viable, and highlight the fact that a tightly-coupled mechanism for bandwidth monitoring and throttling is key to reducing the overheads of the control policy and make it suitable for short-lived tasks.

ACKNOWLEDGMENT

The research presented in this paper has received funding from ECSEL JU projects COMP4DRONES (GA No 826610) and AI4CSM (GA No 101007326). The authors would like to thank their colleague Luca Miccio for his help with the experimental setup and results.

REFERENCES

[1] [n. d.]. Solving Multicore Interference for Safety-Critical Applications. https://www.ghs.com/download/whitepapers/GHS_multicore_interference.pdf

[2] I. Agirre, J. Abella, M. Azkarate-Askasua, and F. J. Cazorla. 2017. On the tailoring of CAST-32A certification guidance to real COTS multicore architectures. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 1–8. https://doi.org/10.1109/SIES.2017.7993376

[3] Giovanni Agosta, William Fornaciari, Giuseppe Massari, Anna Pupykina, Federico Reghenzani, and Michele Zanella. 2018. Managing Heterogeneous Resources in HPC Systems. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms* (Manchester, United Kingdom) *(PARMA-DITAM '18)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3183767.3183769

[4] Ahmed Alhammad and Rodolfo Pellizzoni. 2014. Schedulability analysis of global memory-predictable scheduling. In *Proceedings of the 14th International Conference on Embedded Software*. 1–10.

[5] Ahmed Alhammad and Rodolfo Pellizzoni. 2014. Time-predictable execution of multithreaded applications on multicore systems. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[6] Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. 2015. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 285–296.

[7] W. Ali and H. Yun. 2017. Work-In-Progress: Protecting Real-Time GPU Applications on Integrated CPU-GPU SoC Platforms. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 141–144. https://doi.org/10.1109/RTAS.2017.26

[8] Stanley Bak, Gang Yao, Rodolfo Pellizzoni, and Marco Caccamo. 2012. Memory-aware scheduling of multicore task sets for real-time systems. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 300–309.

[9] Antoine Blin, Cédric Courtaud, Julien Sopena, Julia Lawall, and Gilles Muller. 2016. Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. 109–119. https://doi.org/10.1109/ECRTS.2016.18

[10] R. Cavicchioli, N. Capodieci, and M. Bertogna. 2017. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 1–10. https://doi.org/10.1109/ETFA.2017.8247615

[11] Roberto Cavicchioli, Nicola Capodieci, Marco Solieri, Marko Bertogna, Paolo Valente, and Andrea Marongiu. 2020. Evaluating Controlled Memory Request Injection to Counter PREM Memory Underutilization. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 85–105.

[12] Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J. Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. 2020. Multi-Core Devices for Safety-Critical Systems: A Survey. *ACM Comput. Surv.* 53, 4, Article 79 (Aug. 2020), 38 pages. https://doi.org/10.1145/3398665

[13] Francesco Conti, Daniele Palossi, Andrea Marongiu, Davide Rossi, and Luca Benini. 2016. Enabling the heterogeneous accelerator model on ultra-low power microcontroller platforms. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1201–1206.

[14] C. Courtaud, J. Sopena, G. Muller, and D. Gracia Pérez. 2019. Improving Prediction Accuracy of Memory Interferences for Multicore Platforms. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. 246–259.

[15] B. Forsberg, L. Benini, and A. Marongiu. 2018. HePREM: Enabling predictable GPU execution on heterogeneous SoC. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 539–544. https://doi.org/10.23919/DATE.2018.8342066

[16] Björn Forsberg, Luca Benini, and Andrea Marongiu. 2021. HePREM: A Predictable Execution Model for GPU-based Heterogeneous SoCs. *IEEE Trans. Comput.* 70, 1 (2021), 17–29. https://doi.org/10.1109/TC.2020.2980520

[17] Jan Kiszka, , and other community contributors. 2020. Jailhouse: Linux-based partitioning hypervisor. Siemens. https://github.com/siemens/jailhouse.

[18] Zoltan Majo and Thomas R Gross. 2011. Memory management in NUMA multicore systems: trapped between cache contention and interconnect overhead. In *Acm Sigplan Notices*, Vol. 46. ACM, 11–20.

[19] Renato Mancuso, Roman Dudko, and Marco Caccamo. 2014. Light-prem: Automated software refactoring for predictable execution on cots embedded systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 1–10.

[20] Joel Matějka, Björn Forsberg, Michal Sojka, Přemysl Šůcha, Luca Benini, Andrea Marongiu, and Zdeněk Hanzálek. 2019. Combining PREM compilation and static scheduling for high-performance and predictable MPSoC execution. *Parallel Comput.* 85 (2019), 27 – 44. https://doi.org/10.1016/j.parco.2018.11.002

[21] Larry W McVoy, Carl Staelin, et al. 1996. lmbench: Portable Tools for Performance Analysis.. In *USENIX annual technical conference*. San Diego, CA, USA, 279–294.

[22] Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. 2014. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *2014 26th Euromicro Conference on Real-Time Systems*. 109–118. https://doi.org/10.1109/ECRTS.2014.20

[23] Ignacio Sañudo Olmedo, Nicola Capodieci, Jorge Luis Martinez, Andrea Marongiu, and Marko Bertogna. 2020. Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 213–225. https://doi.org/10.1109/RTAS48715.2020.000-5

[24] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. 2011. A predictable execution model for COTS-based embedded systems. In *RTAS'11*. IEEE.

[25] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. 2010. Worst case delay analysis for memory interference in multicore systems. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 741–746.

[26] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. 2017. Look mum, no VM exits!(almost). *arXiv preprint arXiv:1705.06932* (2017).

[27] Muhammad R. Soliman and Rodolfo Pellizzoni. 2019. PREM-Based Optimal Task Segmentation Under Fixed Priority Scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 133)*. 4:1–4:23. https://doi.org/10.4230/LIPIcs.ECRTS.2019.4

[28] The Ohio State University. 2011. PolyBench/C the Polyhedral Benchmark suite. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/. Online; accessed 14 April 2022.

[29] Bogdan Marius Tudor, Yong Meng Teo, and Simon See. 2011. Understanding off-chip memory contention of parallel programs in multicore systems. In *2011 International Conference on Parallel Processing*. IEEE, 602–611.

[30] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. 2016. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 1–12. https://doi.org/10.1109/RTAS.2016.7461361

[31] Pirmin Vogel, Andrea Marongiu, and Luca Benini. 2015. An Evaluation of Memory Sharing Performance for Heterogeneous Embedded SoCs with Many-Core Accelerators. In *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores, COSMIC@CGO*. 6:1–6:9. https://doi.org/10.1145/2723772.2723775

[32] Hao Wen and Zhang Wei. 2017. Interference Evaluation In CPU-GPU Heterogeneous Computing. In *IEEE High Performance Extreme Computing Conference (HPEC)*.

[33] Shinichi Yamagiwa and Koichi Wada. 2009. Performance study of interference on gpu and cpu resources with multiple applications. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–8.

[34] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. 2012. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems* 48, 6 (2012), 681–715.

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

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

[37] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 2013. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Techn. and Appl. Symp. (RTAS)*. IEEE.