Evaluating Controlled Memory Request Injection to Counter PREM Memory Underutilization

Roberto Cavicchioli, Nicola Capodieci, Marco Solieri,
Marko Bertogna, Paolo Valente, and Andrea Marongiu

University of Modena And Reggio Emilia, Italy
Dept. of Physics, Informatics and Mathematics
{name.surname}@unimore.it

Abstract. Modern heterogeneous systems-on-chip (HeSoC) feature high-performance multi-core CPUs tightly integrated with data-parallel accelerators. Such HeSoCS heavily rely on shared resources, which hinder their adoption in the context of Real-Time systems. The predictable execution model (PREM) has proven effective at preventing uncontrolled execution time lengthening due to memory interference in HeSoC sharing main memory (DRAM). However, PREM only allows one task at a time to access memory, which inherently under-utilizes the available memory bandwidth in modern HeSoCs. In this paper, we conduct a thorough experimental study aimed at assessing the potential benefits of extending PREM so as to inject controlled amounts of memory requests coming from other tasks than the one currently granted exclusive DRAM access. Focusing on a state-of-the-art HeSoC, the NVIDIA TX2, we extensively characterize the relation between the injected bandwidth and the latency experienced by the task under test. The results confirm that for various types of workload it is possible to exploit the available bandwidth much more efficiently than standard PREM arbitration, often close to its maximum, while keeping latency inflation below 10%. We discuss possible practical implementation directions, highlighting the expected benefits and technical challenges.

Keywords: heterogeneous systems-on-chip · Memory interference · Predictable Execution

1 Introduction

In recent years, embedded systems designs have been increasingly embracing the heterogeneous system-on-chip paradigm (HeSoC), where several and possibly non-identical general-purpose CPUs are coupled to various accelerators (e.g., GPUs for general purpose computing, Digital Signal Processors, etc.), achieving high processing capacity at a comparatively low cost. While these systems are capable of achieving very high GOps/W targets, they are designed for optimal best-effort performance, not at all for timing predictability. Designing a predictable system requires characterizing its schedulability, i.e., formally assessing whether the timing constraints of each work unit (task) can be satisfied with the available computing resources (CPU cores, accelerator cores, memory). In the real-time literature [2], applications are generally composed of tasks characterized by a *period* (minimum time span between two instances of the same task),

a Worst-Case Execution Time ($WCET$) and a *deadline*. A taskset is *schedulable* if there exists a mapping of its tasks to the available compute/memory resource that does not cause a deadline miss at runtime.

In HeSoCs, CPU cores or other computing units can concurrently access shared memory resources such as caches and system DRAM. The best-effort, throughput-oriented arbitration mechanisms of such on-chip shared resources – which increasingly becomes a point of contention, severely impacting the execution time of concurrent tasks – coupled to the often undisclosed nature of their internal working poses severe challenges to the adoption of HeSoCs in the context of real-time applications. Real-time scheduling has traditionally focused on scheduling CPU computation, assuming that the Worst-Case Execution Time can be computed for each task running in the system. However, when considering different applications running concurrently on a modern HeSoC, with several actors featuring different bandwidth usage, the measured WCET can vary significantly, depending on the global system schedule.

The Predictable Execution Model (PREM) was originally proposed in the context of single-core CPUs [11] to provide robustness to memory-access interference from I/O devices, and was later extended to avoid inter-core interference in multi-core CPUs [1] and in HeSoCs [6]. PREM assumes that tasks are split into *memory* and *compute phases*, with all shared-memory accesses in the memory phases. By scheduling memory phases separately, the system designer has full control on shared-memory interference. In particular, interference can be canceled completely by executing one memory phase at a time.

While this greatly reduces WCET pessimism, the very method for canceling interference entails one of the main drawbacks of PREM: executing only one memory phase at a time implies severe under-utilization of the shared-memory bandwidth in most cases.

In this paper, we conduct a thorough experimental study aimed at assessing the potential benefits of extending PREM with a technique capable of *injecting controlled amounts of memory accesses* coming from other tasks than the one currently granted exclusive access to the memory. Focusing on a state-of-the-art HeSoC, the NVIDIA Tegra X2, we extensively characterize the relation between the injected bandwidth and the memory-access latency experienced by the task(s) under test. The NVIDIA TX2 features a *host* CPU design architected as two *clusters*, one made of four ARM Cortex-A57 and the other made of two DENVER cores, each featuring local, shared L2 cache. On both the Cortex-A57 and the DENVER clusters, if the workload is memory-intensive, *controlled injection* allows reaching over 80% of the maximum cluster bandwidth with virtually no impact on the execution time of the PREM task. If the workload has more sporadic/random accesses to main memory, we can improve over PREM by at least 600%, in some cases reaching maximum exploitation of the available bandwidth.

Based on these observations, we discuss possible practical implementation directions, highlighting the expected benefits and technical challenges.

This paper is organized as follows: Section 2 discusses background notions related to the PREM execution model and the target hardware platform, as well as our benchmarking methodology and the experimental setup. Section 3 presents and discusses the results of the various experiments we conduct to assess the benefits of *controlled injection*. Section 4 discusses possible practical ways to implement and deploy the presented approach in real-life application scenar-

ios. Section 5 presents related works, before concluding the paper in Section 6, highlighting future research directions.

2 Controlled Injection: Background, Experimental Methodology and Setup

The Predictable Execution Model (PREM) is designed to isolate accesses to shared main memory from different actors. The original proposal in [11] was meant to provide robustness to CPU memory-access interference from peripheral devices, and was later extended to avoid inter-core interference in multi-core CPUs [1]. This is achieved by partitioning programs into contention-sensitive memory 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.

Most modern multi-core SoC designs leverage heterogeneity at different levels. Typically, 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, DSPs) or programmable logic (FPGA). 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. Globally, these heterogeneous *clusters* share the last-level cache or the main system memory. The latter is also shared with other acceleration devices. These systems must sustain tremendous bandwidth to the main memory, to satisfy requests coming from many actors. Table 1 shows the effect of inter-core bandwidth sharing on a variety of commercial heterogeneous SoCs, highlighting the portion of the total SoC bandwidth that is used by the *host* CPU cores (the focus of this paper). The rightmost column indicates the nominal main memory bandwidth as reported in the official datasheets (where available). We use the *bw_mem* benchmark from the *LMBench* suite [10] to measure the maximum bandwidth request generated by a single CPU (*host*) core and the maximum aggregated bandwidth requested by all the cores inside the CPU complex[1] For each of the tested SoCs where a total nominal bandwidth value is available (i.e., those featuring a GPU), the comparison with the aggregated CPU bandwidth shows that a significant share of this bandwidth is reserved to other devices than the CPU (the acceleration logic). Focusing on the CPU complex, we observe a significant difference between the maximum bandwidth used by a single core and the aggregated memory bandwidth used by the whole CPU *compute cluster*. Even though the bandwidth request in general does not grow up linearly with the number of cores, it is evident that a single CPU core only consumes a fraction of the bandwidth budget allotted to the CPU complex.

In this scenario, the *one-core-at-a-time* memory arbitration model implied by PREM is bound to poorly fit this increasing gap in bandwidth reservation.

[1] *bw_mem* performs a pointer walk over a >100 MiB buffer, using a stride such that consecutive memory accesses request a different L2 cache line. The test is executed for a predefined time window (3s) to measure the average bandwidth.

Table 1. Baseline and composite bandwidth on recent SoCs.

SoC/Board	CPU core count	Arch	BW from single core [GiB/s]	aggreg CPU BW [GiB/s]	Total BW (nominal) [GiB/s]
NVIDIA Jetson Xavier	8	Carmel (ARM v8.2-A)	18	74	137
NVIDIA Jetson TX2	6	Denver2 + Cortex-A57	12(4)	22	59.7
Intel i7-9700K	8	x86_64	22	27	39.4
Xilinx UltraScale Zynq ZCU 102	4	Cortex-A53 (ARM v8-A)	2.3	6.7	-
Xilinx Zedboard Zynq-7000	2	Cortex-A9 (ARM v7)	0.404	0.505	-

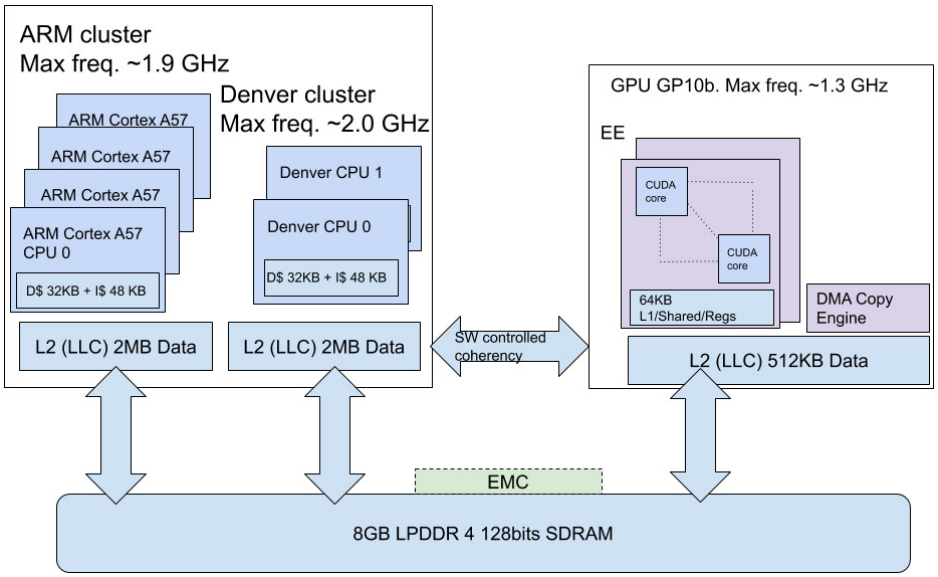

Fig. 1. Block diagram of the Tegra X2 architecture.

PREM could be enhanced with techniques to admit more than one task at a time to access memory. We will explain that 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. We call this technique *Controlled memory-transaction Injection*, proposing a synthetic benchmark that allows evaluating this concept by a thorough characterization of the system.

2.1 Target Architecture

For our benchmarking, and to assess the potential benefits of a *Controlled Injection* scheme, we have selected the NVIDIA Jetson TX2 as our reference HeSoC hardware. The NVIDIA Jetson TX2 is a widely available commercial HeSoC featuring a GPU accelerator governed by a heterogeneous *host* processor.

The *host* is organized in two different *compute clusters*: a quad-core ARMv8 Cortex-A57, and a dual-core ARMv8-compliant *DENVER* processor (NVIDIA proprietary design). Each of the six cores integrates a 32 KiB L1 data cache and

a 48 KiB L1 instruction cache. Furthermore, each *cluster* features a 2 MiB L2 cache shared among its local cores.

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 CPU *clusters* and the GPU accelerator.

In this paper, we focus on the characterization of the *host* memory behavior.

2.2 Benchmarking Methodology

There are many ways to implement *Controlled Injection* (see Section 4 for a discussion on practical solutions). In this paper, our focus is on exhaustively studying the potential benefits of this technique, stressing corner cases via a custom-designed synthetic benchmark, called *mem_bench*[2].

Controlling the access pattern. The first thing we need to be able to control is the type of access pattern our benchmark generates. Traditionally, *sequential* and *random* patterns are adopted for this type of bandwidth/latency measurements, with the former reading memory addresses one word aside (i.e., consecutive words, with unit *stride*) and the latter reading with random *stride*. The NVDIA TX2, like most modern HeSoCs, features a number of hardware blocks aimed at improving the average-case performance (cache prefetchers, DRAM row buffers, etc.). As all these features are platform-specific, for our benchmarking methodology to be general, we need a convenient knob to control the extent to which our workload bypasses such mechanisms, spanning a range of access patterns that goes from purely sequential to fully random (where each memory access really pays the worst-case cost). To achieve this goal, we implement 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.

To model a *sequential* access pattern, we only need to define a $SIZE_{MEM}$ and a *stride* parameter. The pointer walk can then be initialized to read $SIZE_{MEM}$/ *stride* cache lines at that fixed *stride*. In this case, the $SIZE_{MEM}$ is the size of the L2 cache divided by the number of cores, and the *stride* is the size of a cache line (64B). For the *random* access pattern, *mem_bench* accepts two parameters:

1. **the size of the memory portion within the static array from which the addresses to initialize the pointer walk are taken ($SIZE_{MEM}$):** this is key to control the actual distance of the loads/stores (i.e., their cost) when modeling a *random* traffic pattern[3];
2. **the number of cache lines the pointer walk should access ($NLINES$):** this is key to control the size of the PREM memory phase, which, in this work, is assumed to be the L2 cache size divided by the number of cores. As the data is read only once, this parameter dictates the overall duration of the benchmark.

[2] available for download: https://git.hipert.unimore.it/msolieri/membench

[3] Randomly computing the next address inside a memory portion that fits the size of the prefetch buffer generates the same behavior of a *sequential* traffic pattern, as most of the loads/stores will hit in the L2 cache.

The first parameter is in fact used to model the desired traffic pattern mix. Figure 2 shows the results of an experiment where we measure the execution time to read a memory portion of increasing size (on the X axis) on both the Cortex-A57 and the DENVER CPU cores. More specifically, here $SIZE_{MEM} = NLINES{*}SIZE_{cache\_line}$. The bandwidth is computed as $SIZE_{MEM}/exec\_time$.

For both the Cortex-A57 and the DENVER, for small values of $SIZE_{MEM}$, the duration of the transfer is very small, which makes the measurement very sensitive to system overheads. For this reason, the curves initially ramp up, getting closer to their peak value at around 16 and 32KB, respectively. This is the operating point where the use of prefetching mechanisms is still predominant. The more we move to the right, the higher the probability that the strides are wider than the prefetch buffers size and that the requests are difficult to reorder at the DRAM side for better row usage. We can see that for $SIZE_{MEM} \geq 4096$B the traffic pattern generated by our benchmark is truly *random*[4].

Controlling the Injection Rate To evaluate the benefits of *Controlled Injection* as well as to assess the capability of controlling it at a fine granularity, we extend *mem_bench* with an option that allows interleaving a sequence of memory accesses (representative of one PREM task memory phase) with a parametric number of stall cycles, so as to finely tune the injected memory bandwidth request.

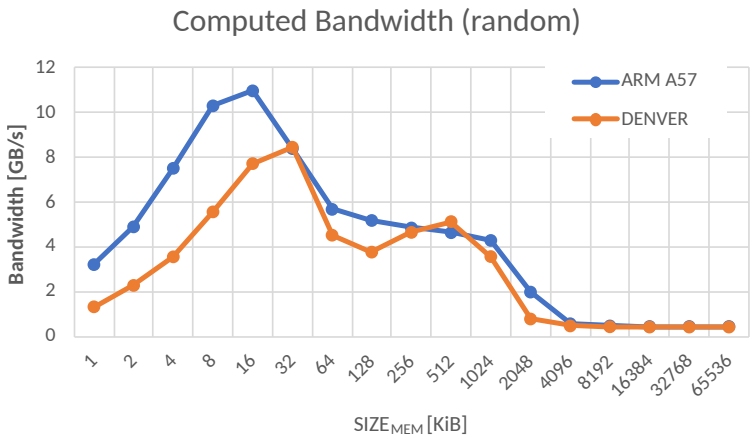

Fig. 2. Bandwidth request for increasingly larger memory portions. Random traffic.

To this aim, we add two more parameters:

1. the number of cache lines (L) that the task can read before being throttled;
2. the number of cycles (C) the task will be throttled for before reading again.

[4] In the remainder of the paper, we use the value $SIZE_{MEM}$ = 12MiB to generate random interference traffic.

L and C define the *duty cycle* of the memory requests issued by the task performing *controlled injection*. In our experimental evaluation, we call *LOAD intensity* the ratio between number of cache lines and the number of throttling cycles. The *LOAD intensity* affects in different ways the *injection rate* that a given task can sustain, based on the type of access pattern the task performs[5]. Intuitively, there are several ways of organizing the memory access pattern so as to obtain a given injection rate. Depending on the granularity at which the injection is done, the generated interference will be different. Our synthetic benchmarking methodology is aimed at capturing the effects of the most fine-grained injection scheme one could conceive, i.e., a single memory access, followed by a controllable number of stall cycles. Evaluating this particular injection scheme (as compared, for example, to coarser grained ones like [21]) in combination with an experimental setup that stresses the most pessimistic operating conditions (see Section 2.3) is in our opinion the fairest way to assess the benefits of *controlled injection*.

mem_bench can be configured to operate in two modes:

1. ***mem_bench_LAT*:** in this mode, the benchmark only **reads once** the $SIZE_{MEM}$ bytes specified as the memory phase. This is the most accurate mode for latency measurements;
2. ***mem_bench_BW*:** in this mode, the benchmark **reads multiple times** the $SIZE_{MEM}$ bytes to ensure the duration (in seconds) of the benchmark exceeds the value specified via an additional $DURATION$ parameter. This is the most accurate mode for bandwidth measurements.

2.3 Experimental setup

Our experiments are aimed at assessing the benefits of *Controlled Injection* as a complementary technique to a standard PREM scheme. In such a system, all the tasks are transformed according to the PREM rules (memory + compute phase), and their memory phases are augmented with the fine-grained *controlled injection* mechanism described in the previous section.

Our target is to exhaustively assess the benefits of this fine-grained *controlled injection* system under the most pessimistic conditions. To model such conditions, we model the PREM taskset as follows: The task under test (UT) is representative of the PREM task currently allowed to access the main memory (in a traditional mutually-exclusive PREM scheduling scheme). The task reads once the amount of data decided by the code generation policy in the compiler: typically the whole cache [5]). The rest of the tasks, which in a standard PREM system would be idle while this UT task is executing, are in our experiments allowed to inject their own memory transactions at varying rates, acting as interfering tasks (IF). In the real PREM+*Controlled Injection* system that we envision, these IF tasks would also be transformed according to the PREM rules: their memory phases would have defined duration and they would be scheduled to maximize their memory bandwidth usage. To model the worst-case interference that the IF tasks could generate, we execute them for the whole time that the UT task is running.

[5] As we will explain later on, the *injection rate* is defined as the ratio between the bandwidth request generated by a task at a given *LOAD intensity* value and the bandwidth request generated by the same task at 100% *LOAD intensity*.

Table 2. Maximum bandwidth request (in MiB) generated by a single Cortex-A57 or DENVER core when executing in isolation (Alone) or with the other CPU complex and GPU generating maximum bandwidth request (All together).

	Alone		All together	
	sequential	random	sequential	random
Cortex-A57	4204	425	3650	400
DENVER	9078	409	8760	395

Specifically, IF tasks start execution before we launch the UT task, and complete execution after the UT task has terminated.

This is representative of an ideal PREM+*Controlled Injection* system, where the interference is constant, because we always have sustained injection traffic and we can schedule it so as to never leave gaps. In a real system, such gaps would be present, so the interference suffered by the UT task would in general be smaller. In this sense, the results we show here are pessimistic with respect to the benefits of *controlled injection*.

Considering the cluster-based nature of the target hardware, we conduct our experiments in three settings:

1. inside the Cortex-A57 CPU cluster;
2. inside the DENVER CPU cluster;
3. across the two compute clusters.

For experiments 1 and 2, we consider a single UT task, 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 *controlled injection* by IF tasks, 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 (and thus the injection rate) from near-zero to 100%.

The Cortex-A57 and the DENVER cores read memory chunks of 512KiB and 1024KiB respectively (the size of their whole L2 cache divided by the number of cores in each cluster). Table 2 shows the measured bandwidth in MiB for a single Cortex-A57 or DENVER core considering both *sequential* and *random* access patterns. The two groups of columns refer to the case when the observed core is the only one that generates memory requests (*Alone*) and to the case when the cores from the other CPU complex and from the GPU are all generating memory requests at full speed (*All together*). We have observed that the interconnects and the memory controller implement static partitioning of the bandwidth between the Cortex-A57 complex, the DENVER complex and the GPU complex. Cores from a given complex are allowed to use some of the bandwidth allocated to a different complex if this is not used, as the difference between the *Alone* and *All together* numbers show. For this reason, while we run our experiments on a target complex, we keep the rest of the complexes active and reclaiming as much as possible of their own bandwidth (e.g., to conduct experiment 1, we keep all the cores from the DENVER and GPU complexes active executing *sequential* access patterns).

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.

To stress all the possible corner cases, we consider various combinations of access patterns. Indeed, even if PREM compilers try hard to generate sequential patterns for their memory phases, random patterns are unavoidable in certain applications [6]. Using the benchmarks described in the previous sections, we generate four different combinations for UT and IF tasks, considering both *sequential* (SEQ) and *random* (RAN) access patterns:

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

For increasing *LOAD intensity/injection rates*, each of our plots shows a breakdown of the bandwidth usage among different cores (stacked areas) and the latency increase experienced by the UT task. In this way, it is easy to appreciate how much *controlled injection* can be tolerated in the various configurations before the UT task significantly suffers from the interference. Note that bandwidth and latency for the UT tasks are measured in different runs of the benchmark: as already mentioned, *mem_bench_LAT* provides the most accurate way of measuring the effect of interference on PREM tasks (which are characterized by relatively small burst transfers), while *mem_bench_BW* provides the most accurate way of measuring how much bandwidth the UT tasks can utilize under interference.

To limit the noise from operating system services (power management, graphics server, etc.), we do the following for all the experiments: (i) we set the maximum operating frequencies on both the CPU clusters, the GPU and the memory controller via the `jetson_clock` command with root privileges; (ii) the operating system (Linux for Tegra (L4T) Ubuntu 18.04) is set to run level 2 via the `telinit` command; and (iii) all the tasks are pinned to a core using the `taskset` command, and their priority is set to -15 using the `nice` command.

For the latencies, we run each experiment 100 times, and we take the worst-case value, filtering out outliers due to OS activities. In all our experiments, outliers can be easily spotted, as their values are one to two orders of magnitude higher than the vast majority of the samples. Quantitatively, in all our experiments less than 5% of the samples were discarded.

3 Evaluation

3.1 Effects of Controlled Injection Within the Cortex A57 Cluster

Figures 3 and 4 show the results for our experiments within the Cortex-A57 compute cluster. The plots show *LOAD intensity* on the bottom X-axis, and the *injection rate* of a single IF task on the top X-axis. This metric is computed as the bandwidth requested by an IF task for a particular *LOAD intensity*, normalized to the maximum bandwidth that the same IF task can request for the same experiment (i.e., when the *LOAD injection* reaches 100%). It represents probably the most significant way of visualizing the degree of *controlled injection* the system is sustaining, which allows for a more direct comparison between

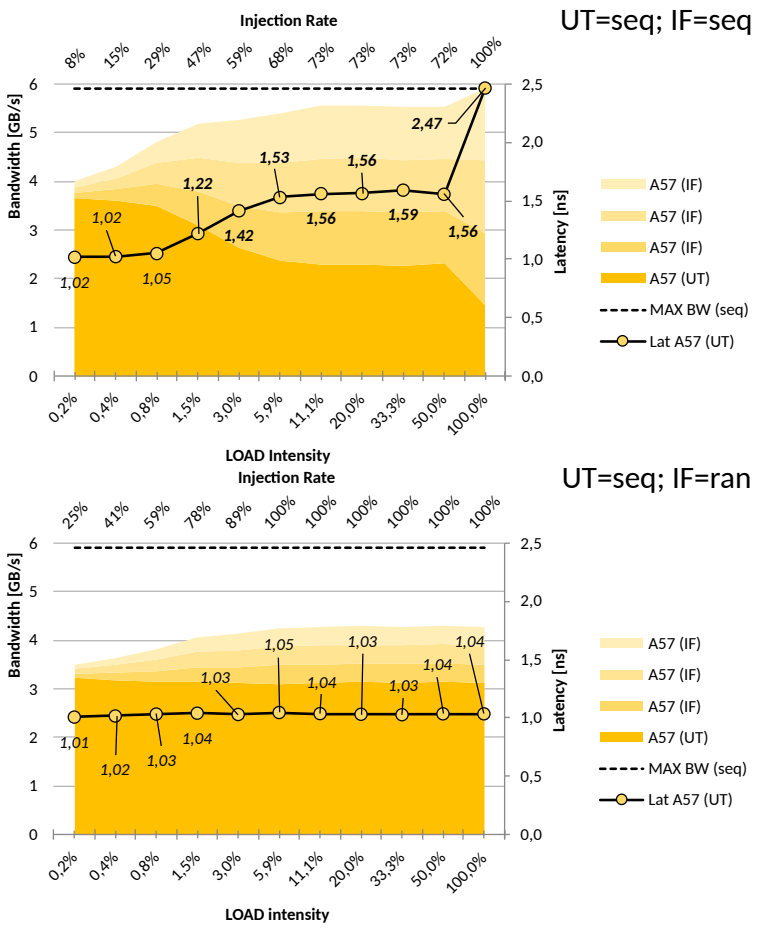

Fig. 3. Effects of controlled injection within the ARM cluster. Task under test has sequential traffic, cache transfer is 512KiB.

bandwidth usage and latency increase. The plots show measured bandwidth on the left Y-axis, and latency on the right Y-axis. The stacked area plots show the breakdown of the total utilized memory bandwidth (to be read on the left Y-axis) by the UT task (the darker area) and the IF tasks. The plots also highlight the measured MAX bandwidth that the four ARM cores can cumulatively request when generating memory transactions at full throttle (for both SEQ and RAN access patterns). The black line with yellow markers shows the increase in execution time (latency, to be read on the right Y-axis) experienced by the UT task while IF tasks are injecting extra memory transactions at increasing rates. More specifically, the values on the markers represent the latency of the test with interference normalized to the baseline execution latency in absence of interference.

If we focus on the bandwidth usage (stacked areas), in all these plots it seems that a significant portion of the available bandwidth, which is not exploited by

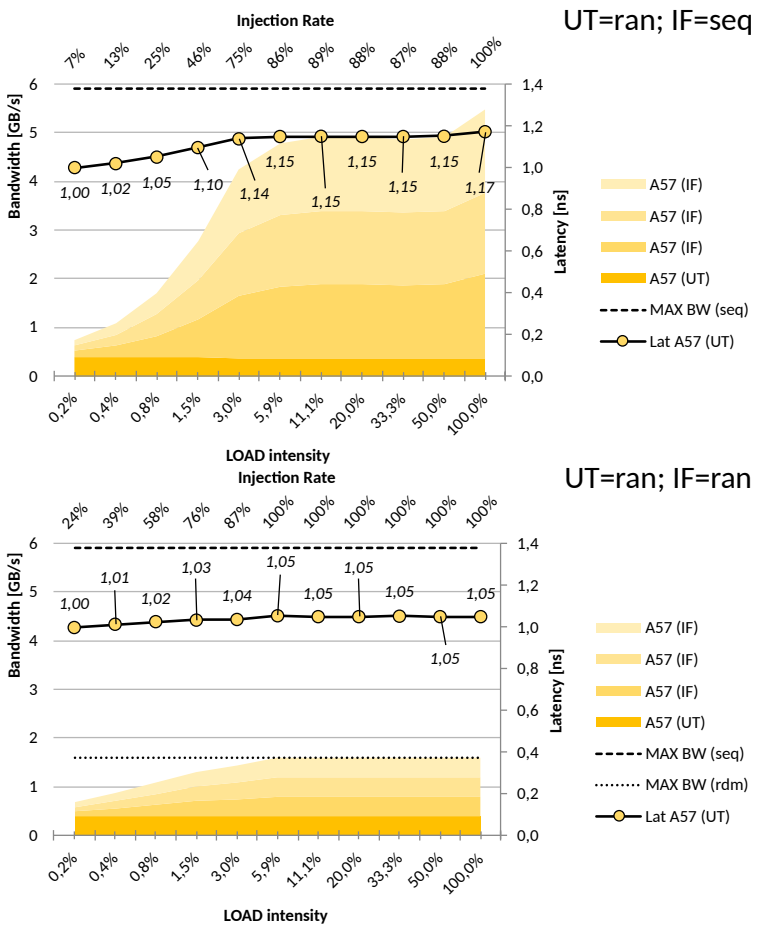

Fig. 4. Effects of controlled injection within the ARM cluster. Task under test has random traffic, cache transfer is 512KiB.

the PREM task alone (as shown when the *injection rate* is at its minimum), can be utilized by *controlled injection* (as shown when we move to the right). By combining the information from the latency curve and the bandwidth areas it is possible to identify the sweet spot where, for each traffic pattern combination, we can gain the most.

Focusing on the first subplot, (UT=SEQ; IF=SEQ), where both the task under test and the interference traffic feature sequential access pattern (i.e., the most sensitive case to interference), we see that the cumulative bandwidth request from all the Cortex-A57 cores is limited to only around 6GB. As already mentioned, the breakdown shows that this is due to system-level bandwidth partitioning, as a single core utilizes 66% of that maximum bandwidth.

If we focus on the latency curve, we notice that a 0.8% *LOAD intensity* by any IF task implies 30% *injection rate*, perturbing the execution time of the UT task by a tiny 5%. Overall, in this case *controlled injection* allows to reach 81%

of the maximum sequential bandwidth with virtually no 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 when 100% *controlled injection* is allowed. This brings a 37% increase in bandwidth usage, overall reaching 72% of the maximum sequential bandwidth. Focusing on a RAN UT task (Figure 4), when the IF task is of type SEQ we see that *controlled injection* has a tremendous effect on increasing memory bandwidth usage. Clearly, the tolerated increase in execution time for the UT task varies from one system (or one application domain) to another, but we can see that even with 100% injection rate the latency never exceeds 17%. In this case we observe a staggering 1365% increase in bandwidth usage, reaching 93% of the maximum sequential bandwidth. If a more conservative 10% is chosen as a maximum tolerated increase in latency, we can see that this is achieved for a *LOAD intensity* of around 1.5% (46% *injection rate*). Even in this case, *controlled injection* allows for a 626% increase in bandwidth usage.

Finally, when both the UT and the IF tasks are of type RAN, we see that the bandwidth requests can be fully summed up, increasing bandwidth usage by 300% without impacting the execution time of the UT task.

3.2 Effects of Controlled Injection Within the DENVER Cluster

Figures 5 and 6 show the results for our experiments within the DENVER compute cluster. The setup is identical to the Cortex-A57 compute cluster, as is the information in the plots and the general observations that can be drawn. The benefits of *controlled injection* in the DENVER cluster are even more pronounced, as one single DENVER core roughly uses half the available bandwidth for the cluster, as can be seen in the UT=SEQ, IF=SEQ plot (whereas in the Cortex-A57 case a single core used 66% of the total). Although the numbers are slightly less stable in this experiment, it can be seen that even when the *injection rate* reaches 100% the increase in the latency of the UT task stays around 10%. The bandwidth usage reaches full efficiency, as it gets doubled.

Similar to the Cortex-A57, 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. Still, *controlled injection* can be applied at full throttle without disturbing the UT task.

When the UT task is of type RAN, there is always to gain from *controlled injection*. If the IF task is of type SEQ, we increase bandwidth usage by 465% while keeping latency increase below 10% (and we could increase it by 2300% if a latency increase of up to 18% could be tolerated). When the IF task is of type RAN, we can double the bandwidth usage without impacting the UT task at all.

3.3 Inter-Cluster Effects of Controlled Injection

After studying the benefits of *controlled injection* 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

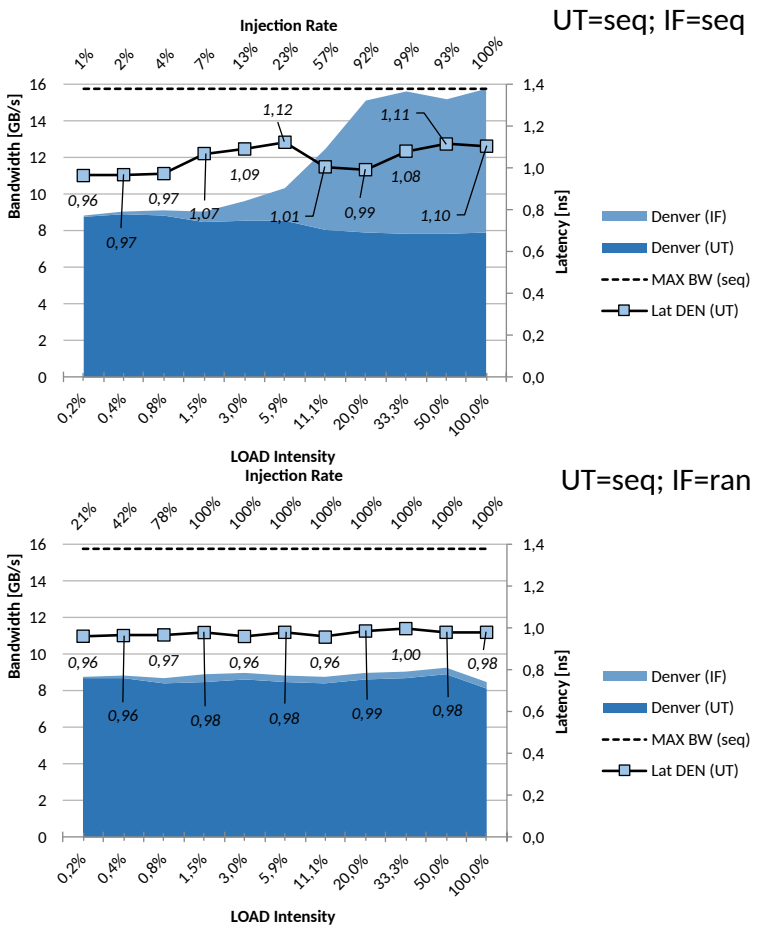

Fig. 5. Effects of controlled injection within the DENVER cluster. Task under test has sequential traffic, cache transfer is 1024KiB.

inactive), and vice-versa. We don't show whole plots for this experiment (which we consider preliminary to what follows), but we report here the most important findings.

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 *injection rate* of the IF tasks running on the other cluster (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 *controlled injection* can effectively achieve.

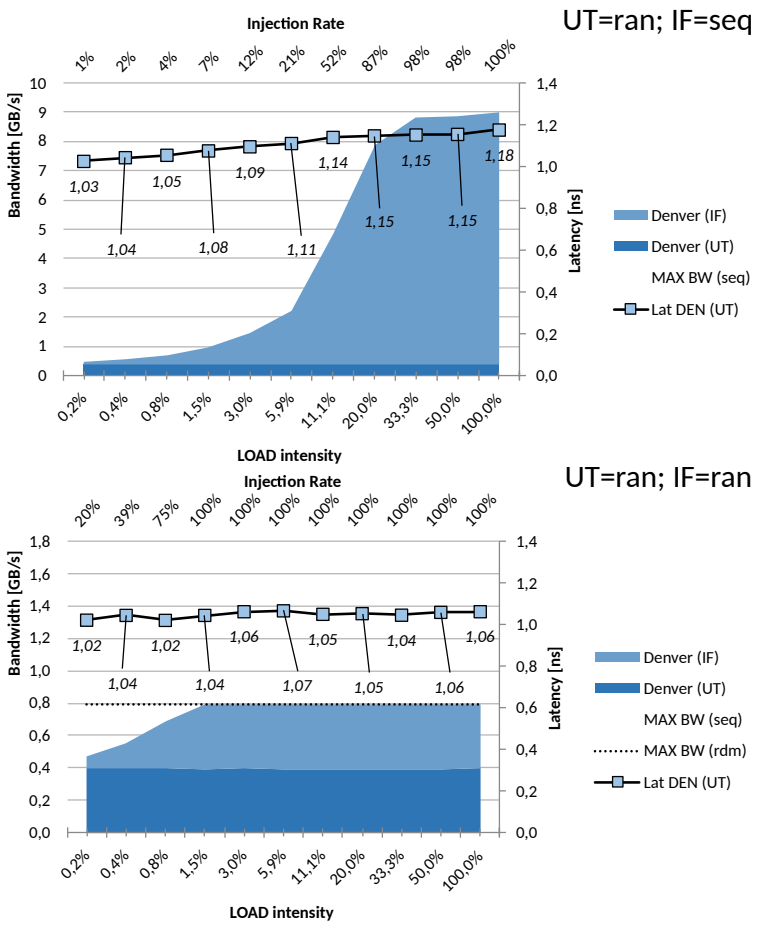

Fig. 6. Effects of controlled injection within the DENVER cluster. Task under test has random traffic, cache transfer is 1024KiB.

Figures 7 and 8 show the results for our last 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 *controlled injection* already seen within each cluster are consolidated. When both the UT and IF tasks generate SEQ requests we can tune IF tasks to inject up to a *LOAD intensity* of 3%, with an increase in the UT tasks latency within 10%. This brings a 35% increase in bandwidth usage compared to only allowing a single Cortex-A57 and a single DENVER core to access memory (a basic PREM scheme). 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 13%. 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

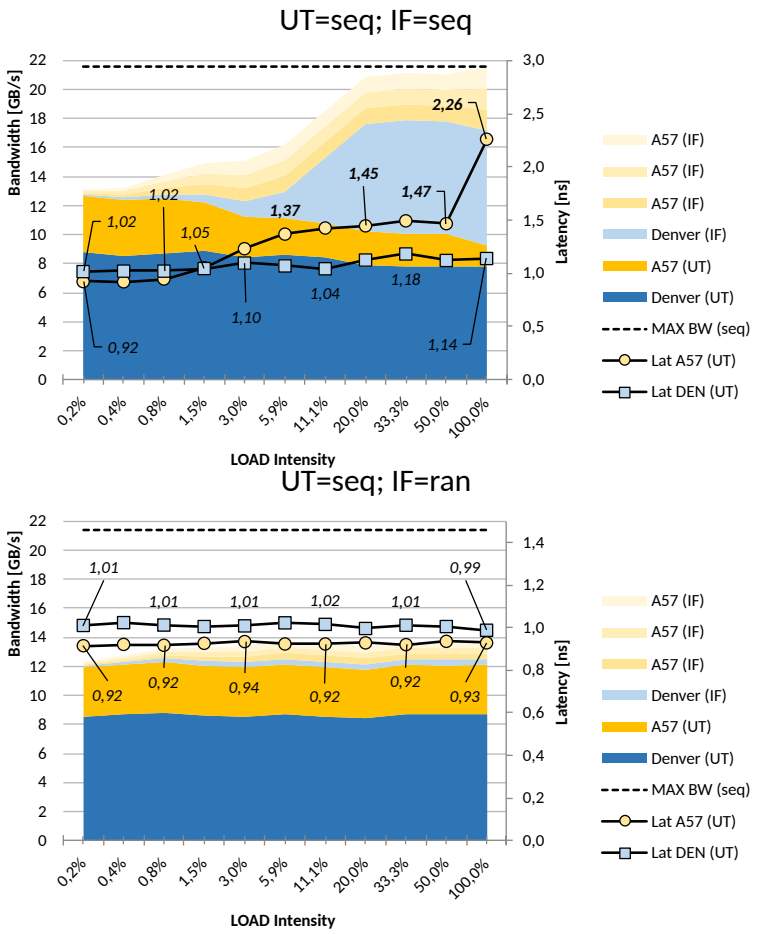

Fig. 7. Effects of controlled injection among the ARM and DENVER clusters (SoC level). Task under test has sequential traffic, cache transfer is 512KiB for the ARM and 1024KiB for the DENVER.

cluster executes (SEQ or RAN) while all cores from the other cluster (and from the GPU) execute SEQ IF tasks. In this particular experiment, we have less interference coming from the other cluster, as one of the cores hosts the UT task. 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 the highest potential for making a better use of the available bandwidth. If the IF tasks are of type SEQ, they can inject at a *LOAD intensity* of up to 3% to keep the latency increase below 10%, improving bandwidth usage by 628% compared to PREM (a single Cortex-A57 and a single DENVER). Note that it is only the Cortex-A57 that poses this limitation. The maximum interference observed on the UT task running on the DENVER core is 11%, when the *LOAD intensity* reaches 100%. At this point, the overall increase in bandwidth usage is 1822%. If the IF tasks are of type

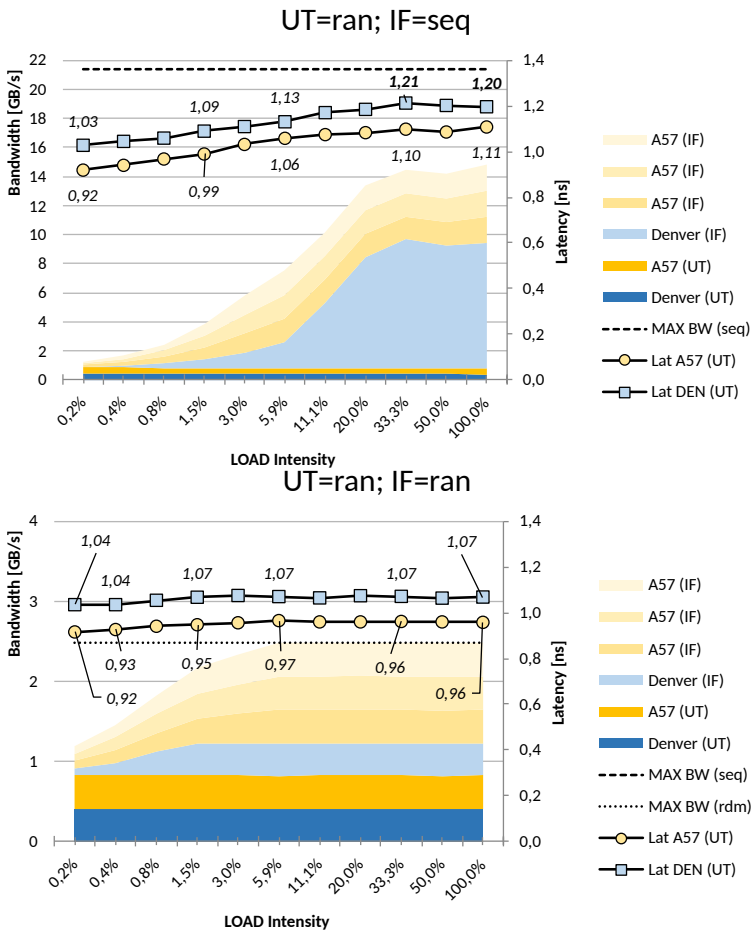

Fig. 8. Effects of controlled injection among the ARM and DENVER clusters (SoC level). Task under test has random traffic, cache transfer is 512KiB for the ARM and 1024KiB for the DENVER.

RAN, they can inject at full throttle, stacking up additively all the bandwidth requests and improving bandwidth usage by 202%.

4 Practical Realization of the Injection Scheme

Implementing *controlled injection* requires design choices that are inherently tight not only to the specific real-time requirements, but also to the particular software stack of interest. Hence, since it would be impossible to provide a definitive reference implementation, we guide the reader along the most important points.

Injection Rate Limit Although effective in increasing the system utilization, bandwidth injection on a PREM-like setup needs to be bounded to preserve the memory access determinism.

Safe approach Given a platform of interest, and an expendable relative latency for memory phases l and characterization of the platform's memory subsystem like the one we presented, it is easy to to find the greatest injection load intensity i such that, for any intensity $i' \leq i$, we measure a relative intensity $l' \leq l$. On a TX2 platform, for instance, if we assume $l = 5\%$, and look at the less favourable worst case for injection SEQ-SEQ, we find $i = 0.8\%$. This practically means that we are gaining: 2.1 GiB/s from 3 cores injecting on A57; 0.3 GiB/s from 1 core injecting on Denver; and 2.7 GiB/s for 3+1 cores injecting on the whole SoC. Although it produces experimental guarantees, this approach may be considered expensive or rigid. Also, it is prone to a tradeoff between bandwidth underutilization and latency degradation, similarly to the m-PREM technique discussed in Sect. 5.

Dynamic approach Alternatively, if a hot measurement of the latency impact can be performed, then such information can be fed to a suitable adaptive algorithm which dynamically increases the injection rate, until reaching an oscillation around the optimal value. In this case, the optimization function has to be constructed or parameterized in order to provide a bounded latency impact that meets the safety requirements.

Injection Control Regardless of the injection rate limitation approach, such limit has to be enforced, when injecting tasks are allowed to execute alongside a PREM task. We discuss two mutually non-exclusive techniques.

Fine-grained control When tasks' code is automatically transformed to adhere to the PREM scheme, a compiler is also part of the PREM runtime [6]. This enables the memory interval compiled code to be regularly interleaved by an idle instruction sequence, whose length can be dynamically determined, or statically hard-coded. In this case, the control on the injection rate is the finest possible, exactly as the idle cycle C used in the empirical section, but we need all tasks to be rebuilt on the custom-compiler.

Transparent control A centralized and secure injection control mechanism can be integrated in the underlying system-layer software, where the PREM admission control also usually resides. For example, an injection server can conveniently be implemented in a hypervisor, to periodically throttle an injecting task, and possibly measure the impacted latencies. Feasibility has been already been proven by the implementation of a memory guarding server [21] for the memory partitioning version [7] of Jailhouse [3], Experiments on Nvidia TX2 showed [13] that such server may run with a period of $16\mu s$ without costing more than 2% time utilization on the injecting task, or even every $1\mu s$, costing the 30%. Although courser-grained than the previous one, this technique is compatible with PREM-foreign code, thus enabling injection on third-party proprietary portion of the software stack (e.g., OS and libraries).

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 [12], HPC-oriented systems [8,15] and even the magnitude on memory interference of co-running integrated and discrete GPUs [4, 18, 19] has been measured in previous work.

PREM was originally proposed for single-core CPUs [11] to provide robustness to memory-access interference from peripheral devices, and was later extended to avoid inter-core interference in multi-core CPUs [1, 14]. After that, there has been extensive work on controlling CPU-GPU interference with PREM on HeSoCs [5,6,9]. PREM splits tasks into memory and compute phases, and schedules memory phases one at a time to prevent interference. While this simple approach allows to greatly reduce the pessimism in WCET estimates, it also heavily underutilizes the memory bandwidth available in modern HeSoCs.
This fact was highlighted by the original authors of PREM [20], which experimentally proved that the latency of main-memory accesses does grow less than linearly with the number of cores accessing memory at the same time. In particular, if each core accesses a different bank, then the latency stays unchanged, independent of the number of competing cores. Based on this observation, the authors define a parameterized algorithm that schedules up to m phases at the same time. The main drawback of this approach is that the user must guess the right value for m for avoiding interference or for keeping it sufficiently low.

This may be particularly tricky if the accesses generated by a task set are not even: for the memory phases of some tasks, a given value of m may be fine, but for other tasks it may be either too low or too high. Even worse, $m = 2$ may be too high for some tasks (or even all tasks), thereby making any utilization boost impossible. Controlled injection, as our evaluation demonstrates, seems able to improve bandwidth utilization also in these unfriendly scenarios, due to its finer granularity.

Memguard [21] provides a different approach to protecting a task's WCET from the adverse effects of memory interference, based on throttling the bandwidth at which every core can access the shared memory, in an attempt to guarantee that each core gets its assigned bandwidth. Although simple and effective, such a mechanism may provide less control on interference compared to PREM, because memory accesses cannot be controlled explicitly. In addition, PREM is a much more general solution, because it is a building block for arbitrary schedules, while throttling is a well-defined control policy. Consider for example a constrained-deadline task[6]. To meet the task's deadlines with Memguard, the task must be assigned a (much) higher bandwidth than that sufficient to complete each instance before the arrival of a new one. This results in (high) bandwidth waste. With PREM, such a task can be scheduled with efficient scheduling algorithms for constrained-deadline tasks, resulting in minimum or even zero bandwidth waste. Previous work has characterized the effects of memory interference on modern HeSoCs [4,16], 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.

[6] i.e., a task with a relative deadline (much) shorter than the minimum inter-arrival time of the task instances.

6 Conclusion and future work

In this paper we presented how, with various types of workload, it is possible to exploit the available bandwidth of HeSoCs in a more efficient way than the canonical PREM arbitration, which conservatively considers main memory as a *one-user-at-a-time* resource. The proposed *controlled injection* technique allows in most practical cases to get close to the maximum available bandwidth, without significantly impacting the latency of the original PREM task. With regards to practical scenarios, authors in [17], for instance, present CAVBench, a collection of applications for autonomous driving vehicles. Profiling these applications leads to the conclusion that such workloads are extremely memory bound, hence the importance of accurately controlling memory bandwidth becomes paramount in latency sensitive scenarios.

As part of our future work, we envision the development of a practical *controlled injection* scheme, built on top of existing PREM implementations that consider both the CPU and the GPU of a HeSoC [5, 6, 9]. *Controlled Injection* also paves the way to new, more effective PREM-based scheduling algorithms. We are currently evaluating scheduling policies where both the UT and the IF tasks are slowed down by the same factor, rather than leaving full bandwidth to a core and allowing the others to inject small amounts of requests.

Acknowledgment

This article is part of a joint collaboration between researchers that work on two projects that have received funding from the European Unions Horizon 2020 research and innovation programme under the ECSEL-JU programme No 826653 (New control) and the POR-FSE 2014-2020 (thematic objective 10) funding assigned by the Regional Authority to the project CUP E96C18001200009.

References

1. Alhammad, A., Pellizzoni, R.: Time-predictable execution of multithreaded applications on multicore systems. In: DATE'14. IEEE (2014)
2. Baruah, S., Bertogna, M., Buttazzo, G.: Multiprocessor scheduling for real-time systems. Springer (2015)
3. Baryshnikov, M.: Jailhouse hypervisor. B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum. (2016)
4. Cavicchioli, R., Capodieci, N., Bertogna, M.: 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). pp. 1–10 (Sep 2017). https://doi.org/10.1109/ETFA.2017.8247615
5. Forsberg, B., Marongiu, A., Benini, L.: GPUguard: Towards supporting a predictable execution model for heterogeneous SoC. In: DATE'17 (2017)
6. Forsberg, B., Benini, L., Marongiu, A.: Heprem: A predictable execution model for gpu-based heterogeneous socs. IEEE Transactions on Computers (to appear)
7. Kloda, T., Solieri, M., Mancuso, R., Capodieci, N., Valente, P., Bertogna, M.: Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 1–14 (April 2019). https://doi.org/10.1109/RTAS.2019.00009

8. Majo, Z., Gross, T.R.: Memory management in numa multicore systems: trapped between cache contention and interconnect overhead. In: Acm Sigplan Notices. vol. 46, pp. 11–20. ACM (2011)
9. Matjka, J., Forsberg, B., Sojka, M., cha, P., Benini, L., Marongiu, A., Hanzlek, Z.: Combining prem compilation and static scheduling for high-performance and predictable mpsoc execution. Parallel Computing **85**, 27 – 44 (2019). https://doi.org/https://doi.org/10.1016/j.parco.2018.11.002
10. McVoy, L.W., Staelin, C., et al.: lmbench: Portable tools for performance analysis. In: USENIX annual technical conference. pp. 279–294. San Diego, CA, USA (1996)
11. Pellizzoni, R., Betti, E., Bak, S., Yao, G., Criswell, J., Caccamo, M., Kegley, R.: A predictable execution model for COTS-based embedded systems. In: RTAS'11. IEEE (2011)
12. Pellizzoni, R., Schranzhofer, A., Chen, J.J., Caccamo, M., Thiele, L.: Worst case delay analysis for memory interference in multicore systems. In: 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010). pp. 741–746. IEEE (2010)
13. Solieri, M., Kloda, T., Bertogna, M., Sojka, M., Baryshnikov, M.: D5.3: Integrated schedulability analysis. Deliverable of the HERCULES project (12 2018)
14. Soliman, M.R., Pellizzoni, R.: 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, pp. 4:1–4:23 (2019). https://doi.org/10.4230/LIPIcs.ECRTS.2019.4
15. Tudor, B.M., Teo, Y.M., See, S.: Understanding off-chip memory contention of parallel programs in multicore systems. In: 2011 International Conference on Parallel Processing. pp. 602–611. IEEE (2011)
16. Vogel, P., Marongiu, A., Benini, L.: 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. pp. 6:1–6:9 (2015). https://doi.org/10.1145/2723772.2723775
17. Wang, Y., Liu, S., Wu, X., Shi, W.: Cavbench: A benchmark suite for connected and autonomous vehicles. CoRR **abs/1810.06659** (2018), http://arxiv.org/abs/1810.06659
18. Wen, H., Wei, Z.: Interference evaluation in cpu-gpu heterogeneous computing. In: IEEE High Performance Extreme Computing Conference (HPEC) (2017)
19. Yamagiwa, S., Wada, K.: Performance study of interference on gpu and cpu resources with multiple applications. In: 2009 IEEE International Symposium on Parallel & Distributed Processing. pp. 1–8. IEEE (2009)
20. Yao, G., Pellizzoni, R., Bak, S., Yun, H., Caccamo, M.: Global real-time memory-centric scheduling for multicore systems. IEEE Transactions on Computers **65**(9), 2739–2751 (Sept 2016). https://doi.org/10.1109/TC.2015.2500572
21. Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., Sha, L.: Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In: Real-Time and Embedded Techn. and Appl. Symp. (RTAS). IEEE (2013)