Journal of Systems Architecture 167 (2025) 103487

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Taking a closer look at memory interference effects in commercial-off-the-shelf multicore SoCs☆

Lorenzo Carletti [a,*], Andrea Serafini [a,b], Gianluca Brilli [a], Alessandro Capotondi [a], Alessandro Biasci [b], Paolo Valente [a], Andrea Marongiu [a]

[a] University of Modena and Reggio Emilia, Via Campi 213/A, Modena, Italy
[b] Evidence S.r.l., Via Pietro Nenni 24, Pisa, Italy

ARTICLE INFO

Keywords:
Multicore System on Chip
Memory interference
Cache

ABSTRACT

Commercial-off-the-shelf (COTS) multicore systems on chip (SoC) represent a cheap and convenient solution for deploying sophisticated workloads in various application domains. The combination of several CPU cores and dedicated acceleration units tightly sharing memory and interconnect systems can provide tremendous peak performance, but also threatens timing predictability due to memory interference. Even when focusing on main CPU cores only, it has been reported that task slowdown due to memory interference can surpass 10×. Such poorly predictable timing behaviors bar greater adoption of COTS multicore SoCs in the domain of timing-critical applications, and motivate the wide activity of the research community to study solutions aimed at mitigating the problem. Understanding worst-case interference patterns on such hardware platforms is fundamental for building any effective memory interference control mechanism. A common assumption in the literature is that worst-case interference is generated by (and therefore assessed through) read-intensive synthetic workloads with 100% cache miss rate. Yet certain real-life workloads exhibit worse slowdown than what is generated under said assumed worst-case, so we study the interference effects of both synthetic and real-life benchmarks on different multicore SoCs. Our experiments indicate that cache thrashing causes the worst interference experienced by real-life benchmarks – due to their different usage of caches – and that there is no universal worst-case workload for every platform.

1. Introduction

Most modern commercial-off-the-shelf (COTS) multicore systems on chip (SoCs) rely on *high parallelism* as a fundamental design paradigm, enabling high performance while maintaining contained power consumption. These systems typically rely on shared main memory (DRAM) and cache hierarchy, and as the number of the on-chip compute units grows contention for these resources intensifies. Under heavy contention scenarios, this causes the tasks executing on the various cores to experience decreased bandwidth and – ultimately – increased execution time (*slowdown*) [1–3].

The problem has been extensively studied before [4–10], and many techniques have been proposed to mitigate its effects and to improve timing predictability [3,11–16]. For any timing analysis or interference mitigation solution to be effective it is important to build on top of models derived from an in-depth understanding of the timing effects that memory interference produces on a given hardware platform. Thus, a detailed characterization of the hardware is the very first step that must be taken when devising solutions to improve its timing predictability.

Synthetic benchmarking methodologies are widely and effectively adopted to this aim [17,18], implementing ad-hoc memory access patterns that stress specific levels of the memory hierarchy. When using synthetic benchmarking, it is often assumed that the worst-case slowdowns are those experienced by read-only or write-only,

☆ This work was supported by the European Union under the NextGenerationEU Programme within the Plan “PNRR - Missione 4 “Istruzione e Ricerca” - Componente C2 Investimento 1.1 “Fondo per il Programma Nazionale di Ricerca e Progetti di Rilevante Interesse Nazionale (PRIN)” by the Italian Ministry of University and Research (MUR)”. Project Title: “Simplifying Predictable and energy-efficient Acceleration from Cloud to Edge (SPACE)”, Project code: 202254AM7H, CUP: E53D23007810006, MUR D.D. financing decree n. 959, 30th June 2023.

* Corresponding author.
E-mail addresses: lorenzo.carletti@unimore.it (L. Carletti), andrea.serafini@unimore.it (A. Serafini), gianluca.brilli@unimore.it (G. Brilli), alessandro.capotondi@unimore.it (A. Capotondi), a.biasci@evidence.eu.com (A. Biasci), paolo.valente@unimore.it (P. Valente), andrea.marongiu@unimore.it (A. Marongiu).

https://doi.org/10.1016/j.sysarc.2025.103487
Received 7 February 2025; Received in revised form 24 April 2025; Accepted 1 June 2025
Available online 4 July 2025

memory-intensive workloads that completely bypass the cache hierarchy (i.e., that generate 100% miss rate). The intuition is that when using this type of workload for both the core *under test* (i.e. the core executing the workload that is subject to the slowdown being measured) and the cores that are generating interference the system is subject to maximum contention (worst-case scenario) [5,6,10,19–21]. However, the massive degree of parallelism and architectural optimizations present in modern multicore CPUs may jeopardize this simplistic assumption, leading to inaccurate timing analysis and, in turn, incorrect memory interference mitigation techniques. This motivated a more in-depth study of the memory interference effects that parallel tasks are subject to when running simultaneously on the CPU cores of modern multicore SoCs.

In this work, we provide a detailed characterization of the memory interference effects generated at various levels of the shared memory hierarchy by different synthetic and real-world workloads [22] co-scheduled on CPU cores. The evaluation, performed on two representative hardware platforms: *Nvidia TX2* and *Xilinx ZU9EG*, indicates that there is no universal worst-case memory access pattern, as the effects induced and suffered slowdowns strongly vary with the hardware. Moreover, the evaluation shows that memory interference happens differently at each level of the shared memory hierarchy, including DRAM-interference effects, *cache trashing*, and all the way down to micro-architectural phenomena.

The rest of the paper is organized as follows: Section 2 introduces the background knowledge underlying the problem of memory interference, the state-of-the-art techniques used for its characterization and the reference hardware platforms used in the experiments. Section 3 extends the characterization to real benchmarks, showing that the read-only synthetic benchmark – typically considered as the worst-case in terms of sensitivity to memory interference – is not the workload that experiences the worst slowdown. Here we also provide examples of how using the wrong synthetic benchmark for interference generation may lead to a wrong characterization of the worst-case scenario. Section 4 explains with additional experiments how cache effects can cause certain less memory intensive tasks to be more subject to interference than others. Section 5 describes related work to ours, positioning our contribution with respect to the vast existing literature. Section 6 provides a discussion on the obtained results and concluding remarks.

2. Background

Fig. 1 shows a generic block diagram of a modern *commercial off-the-shelf* (COTS) multicore SoC (MSoC), highlighting its three main architectural blocks: *CPU complex*, *Accelerator complex*, and *Main Memory*.

The CPU complex is typically composed of multiple cores organized in homogeneous *clusters*, internally sharing part of the memory hierarchy and other interconnect resources. Most modern MSoC also include one or more heavily data-parallel accelerators (GPU, FPGA, DSP) which typically also share main memory with the CPU complex. The main memory is usually implemented as *Dynamic Random Access Memory* (DRAM), and is accessed via a scalable main system interconnect (e.g., a crossbar, or a network-on-chip). As the DRAM access latency spans hundreds of CPU cycles, the CPU complex of a modern MSoC relies on multi-level cache hierarchies to lower the average duration of load/-store operations. Each CPU core has private L1 data and instruction caches; all the cores in a *cluster* share a unified L2 cache; multiple clusters share unified L3 cache. The depth and complexity of the cache hierarchy varies with the number of cores and their clustering. SoCs with a single cluster typically do not require L3 cache, whereas multi-cluster SoCs might feature a fourth level in the cache hierarchy. The cache block at the deepest level is indicated as the Last Level Cache (LLC). Load/store operations that *miss* in the LLC are routed to the main memory. An analogous organization of the internal memory hierarchy is typically found also inside the Accelerator complex.

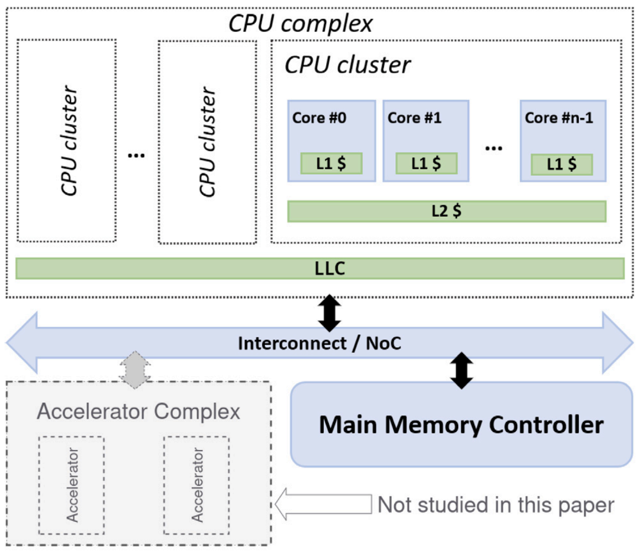

Fig. 1. Shared memory SoC template.

The main interconnect and main memory controller are designed in such a way that **in the common case** there is sufficient *available bandwidth* (BW = bytes/second) to satisfy the read/write requests coming from the various actors (CPU cores, accelerator cores, DMAs, etc.). When several actors (or, **in the worst case**, all of them) access the DRAM simultaneously the *requested bandwidth* might surpass the *available bandwidth*. In such a case, the DRAM is unable to serve the requests coming from all the actors at the speed they are issued. This phenomenon is commonly referred to as *DRAM bandwidth saturation*. This is an important concept for our work: when a task encounters load-/store operations that miss in the LLC while the DRAM bandwidth is saturated, such operations are delayed. The task perceives this effect as a reduced *experienced bandwidth* (a given amount of data is transferred between the hosting CPU and the DRAM in a longer-than-usual amount of time), which ultimately manifests as a slowdown in execution time.

It must be noted that *memory interference* does not only manifest at the DRAM level. Private L1 caches are immune to interference, as they are not shared among cores. L2 or lower-level caches shared by more than one core (from one or multiple clusters) may be subject to interference. Tasks can experience slowdown because of cache interference for different reasons. Loads/stores from different cores and targeting different addresses can be mapped on the same shared cache line. This causes the latest request to *evict* the previous content of the line. Consecutive accesses to the evicted lines will result in a cache miss, hence the slowdown. More specifically, in case of a **load** operation the miss implies a *lookup* and a *refill* for the target address in lower (and slower) levels of the cache hierarchy. In case of a **store** operation the miss still implies the *lookup*, but the slowdown experienced by the task varies based on the write miss policy. If *writethrough* is adopted the task experiencing the miss is stalled until the store is completed. If *writeback* is adopted the store to the DRAM is delayed until the next *eviction*. The slowdown for the writeback is thus experienced by the task that causes this next *eviction*.

Most modern COTS MSoC adhere to the described architectural template, featuring both FPGA-based and GPU-based accelerators. Representative examples of such MSOCs are *Xilinx Zynq Ultrascale+ (ZU9EG)* and *Versal ACAP*, as well as *Nvidia Tegra* SoCs like *Jetson TX2* and *Xavier AGX*.

Table 1 reports the number and the size of the clusters featured by each of these MSoCs, along with the size of their LLC and their write-miss caching policy (writeback for all of them). We conducted a

Table 1
Relevant hardware information for our reference platforms.

	Xilinx ZU9EG	*Versal ACAP VCK190*	*Nvidia TX2*	*Xavier AGX*
CPU cores per cluster	4	2	4	2
Number of clusters	1	1	1	4
Last Level Cache size	1 MB	1 MB	2 MB	4 MB
Cache write-miss policy	writeback	writeback	writeback	writeback

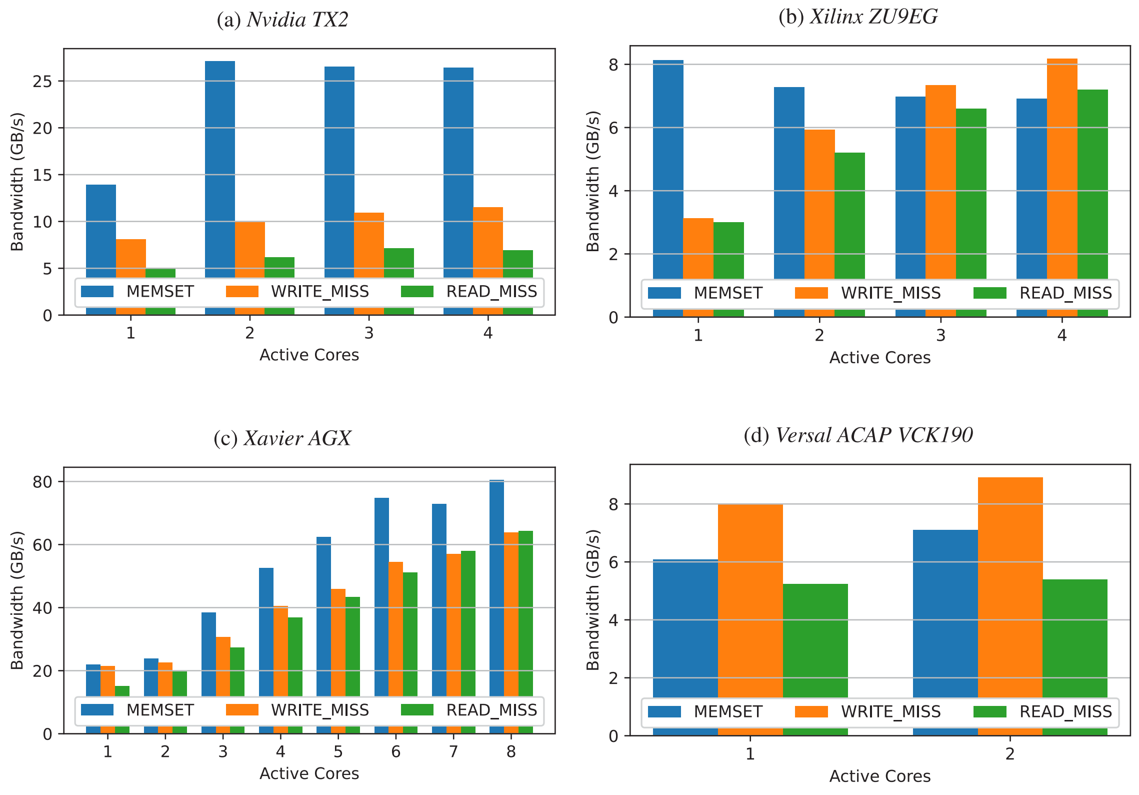

Fig. 2. DRAM bandwidth reached by the three traffic types on our reference SoCs.

preliminary experiment with each of these platforms, aimed at studying DRAM bandwidth saturation under three different types of traffic (for more details on these workload we refer the reader to the next section):

- *READ_MISS*: traffic composed of only **load** operations that miss in LLC, that trigger only cache refills.
- *WRITE_MISS*, traffic composed of only **store** operations that miss in LLC. Consecutive stores target non-contiguous addresses which are (a multiple of) the cache line size far apart. This traffic triggers both cache refills (because the store only writes part of the cache line, the rest needs to be read from memory) and writebacks (upon cache line eviction).
- *MEMSET*: traffic composed of only **store** operations that miss in LLC. Consecutive stores target contiguous addresses, thus entire cache lines are written consecutively. This not only avoids the cache refills from *WRITE_MISS* traffic, but bypasses the cache altogether, enabling the *read-allocate mode* [23]. When this mode is activated, the full set of stores is posted into a *store buffer*, that is asynchronously copied to DRAM, and the contents of the LLC remain unchanged. For the rest of the paper, we refer to this type of traffic as *write-no-allocate* traffic (a name which better reflects the type of memory accesses generated).

Fig. 2 shows the results of the experiment. The X axis shows the number of cores involved and the Y axis the cumulative bandwidth requested (average of 20 executions), for each of the three traffic types. For the *Nvidia TX2* and *Xilinx ZU9EG* platforms it is possible to observe bandwidth *saturation* effects in the plots, as the requested bandwidth reaches a plateau for a number of active cores smaller than the total. Depending on the hardware, different traffic types can either reach saturation with different configurations (e.g., *MEMSET* in *Xilinx ZU9EG* reaches saturation with one single core, whereas the other two traffic types saturate as more cores are involved), or exhibit distinct saturation bandwidths for different traffic types (e.g., *MEMSET* in *Nvidia TX2* has a much higher maximum bandwidth than the other traffic types). It also appears clear that **the workload which generates the highest bandwidth is different for the various platforms**.

On the other hand, the *Versal ACAP VCK190* and the *Xavier AGX* platforms do not exhibit the same clear saturation effects, nor they highlight other interesting phenomenon. The reason for this is that the overall bandwidth is overdimensioned for the total number of cores and number of cores per cluster. We believe however that *Nvidia TX2* and *Xilinx ZU9EG* clustering are more representative of what future SoCs will implement when scaling to larger core counts. This intuition is confirmed by looking at how NVIDIA organized the architecture of *Orin AGX* [24], the successor of *Xavier AGX*. Here both the 12-core and 8-core variants of the SoC organize their cores in 3 and 2 clusters that closely resemble the ones found in *Nvidia TX2* and *Xilinx ZU9EG*.

For this reason in the remainder of the paper we just report our finding on this two platforms, omitting what we observed on *Versal ACAP VCK190* and the *Xavier AGX*, that does not add to the conclusion. Also, in the remainder of the paper we focus on interference effects generated when the CPU complex alone is active. The study of CPU/accelerator interference falls beyond the scope of this paper, and is left as future work.

2.1. Workloads

In this subsection, the workloads used in all the experiments in this paper are described.

2.1.1. Synthetic benchmarking

Synthetic memory-access generators are commonly used to produce controlled interference and measure resulting slowdowns [5,6,10,19–21]. They allow a wide spectrum of access patterns to be easily generated. In general, a synthetic benchmark [17,18] typically has the following parameters (which can be configurable or preset depending on how the benchmark is designed):

1. **Type of memory operation** (t), including *read* (i.e., *loads*), *write* (i.e., *stores*), *r/w* (both *loads* and *stores*). In particular, we differentiate between full *cache line* stores (w, used by workloads of the *MEMSET* type, where stores at contiguous addresses are issued in a loop which fills the cache line) and non-full *cache line* stores (ws, used by workloads of the *WRITE_MISS* type, represented by a single store) due to the significantly different behavior they present;
2. **Number of memory and CPU operations** ($mops, cops$). The synthetic benchmark issues *mops* memory operations of the specified type *t* in a loop. Between one memory operation and the next, *cops* compute operations (or simple no-ops) are executed. This allows to control memory-to-compute ratio and, in turn, to generate more or less interfering (or sensitive-to-interference) workloads.
3. **Address stride** (st). The distance between two consecutive memory accesses (i.e., the *stride*). It heavily influences the cache miss rate (i.e., the sensitivity to interference), as certain specific *stride* configurations may be chosen to avoid the effects of *line prefetching*, depending on what the benchmark wants to assess;
4. **Access pattern** (p). Most real-life workloads tend to generate *sequential* access patterns — which consist of constant, small *strides* [19,25], which the DRAM controller is optimized to handle faster. In contrast, a *random* access pattern is one where the *stride* is updated randomly for any two consecutive accesses, which the DRAM controller takes longer to handle. Note that very long constant strides behave as random patterns [19,25];
5. **Memory footprint** (fp), the *footprint* of the data structure accessed by the benchmark heavily influences cache behavior, as data that fits entirely in cache is bound to generate a high number of *hits*, as opposed to data that exceeds the size of the cache.

Algorithm 1 describes the synthetic benchmark we designed for this paper [26]. The algorithm takes as inputs: (i) the base address addr of the data structure on which the memory operations are performed; (ii) the type t and number $mops$ of memory operations to be performed on the data structure; (iii) the number $cops$ of compute operations to be done after every memory operation; (iv) the stride st between the addresses of any two consecutive memory operations; (v) the global footprint fp of the data structure. The main procedure consists of a loop that iterates $mops$ times. At each iteration it first executes the designated memory operation (specified by the type parameter t) at target address $addr+offset$. After that, the $offset$ is incremented according to the stride parameter st. Finally, a loop that executes $cops$ compute operations is executed.

Algorithm 2 describes the actual synthetic benchmarks *READ_MISS*, *WRITE_MISS* and *MEMSET*. The three synthetic benchmarks call $reps$ times inside of a loop SYNTH_BENCH, with preset parameters. These are the same for all the benchmarks, aside from the type parameter t, which is different between the three (*READ_MISS* has $t = r$, *WRITE_MISS* has $t = ws$, and *MEMSET* has $t = w$).

Algorithm 1: Synthetic Benchmark. LINE_SIZE is a const representing the LLC line size on the target.

```
Function SYNTH_BENCH(addr,t,mops,cops,st,fp)
    Input: $addr$: read or write base address, $t$: access type
           (r/w/rw), $mops$: number of memory operations, $cops$:
           number of cpu operations, $st$: access stride, $fp$:
           memory footprint
    $i \leftarrow 0$;
    $offset \leftarrow 0$;
    while $i < mops$ do
        if $t = r$ then
            load($addr + offset$)
        end
        if $t = w$ then
            for $i \leftarrow 0$ to $LINE\_SIZE$ by 4 do
                store($addr + offset + i$)
            end
        end
        if $t = ws$ then
            store($addr + offset$)
        end
        $offset \leftarrow (offset + st) \mod fp$;
        $i \leftarrow i + 1$;
        $j \leftarrow 0$; while $j < cops$ do
            $j \leftarrow j + 1$;
        end
    end
```

Algorithm 2: Different synthetic benchmarks configurations used for the evaluation.

```
Function READ_MISS($src, reps$)
    Input: $src$: read base address
    for $i \leftarrow reps$ do
        SYNTH_BENCH($src, r, mops, cops, st, fp$)
    end
Function WRITE_MISS($dst, reps$)
    Input: $dst$: write base address
    for $i \leftarrow reps$ do
        SYNTH_BENCH($dst, ws, mops, cops, st, fp$)
    end
Function MEMSET($dst, reps$)
    Input: $dst$: write base address
    for $i \leftarrow reps$ do
        SYNTH_BENCH($dst, w, mops, cops, st, fp$)
    end
```

Many works in the state of the art [5,6,10,19–21] have operated under the assumption that the worst-case access can be modeled by: (i) choosing $t = r$ or $t = ws$ (i.e., instantiating *READ_MISS* or *WRITE_MISS* traffic types); (ii) making the *stride* an integer multiple of the LLC line size with a *sequential access pattern*; (iii) choosing a *footprint* bigger than the full LLC size, as it removes the possibility of one of the cache levels intercepting the memory accesses with cache hits when the algorithm executes in a loop, thus causing a 100% miss rate (meaning DRAM operations). In this paper we further study the problem of worst-case interference characterization, and we show that there are a number of effects which are not correctly captured by the synthetic benchmark generated using what was previously thought as the worst-case access model. This is important, as it means that the real worst-case interference can have a much higher impact on timing than previous work was based on.

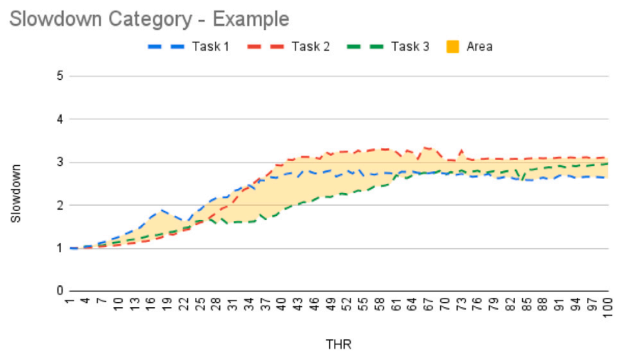

Fig. 3. Example of how the slowdown of multiple Polybench benchmarks combines to create the area of one category in Fig. 4.

2.1.2. The Polybench-ACC benchmark suite

Synthetic benchmarks enable precise control of the generated memory traffic and, presumably, interference. However, the memory access patterns generated by synthetic benchmarks are limited to a subset of the ones generated by real-world workloads, and specific memory interference effects may not be triggered by them. For this reason, we also use real-life benchmarks in this paper, the Polybench-ACC benchmark suite [22]. The suite is a collection of compute kernels that are commonly found inside larger programs belonging to different categories: data mining, stencils, medley and linear-algebra kernels and solvers. In our experimental setup we compile the suite for single-core execution with the default dataset size. For most benchmarks that corresponds to the STANDARD_DATASET size. For benchmark `correlation` the default setting corresponds to EXTRALARGE_DATASET. For `convolution-3d` and `convolution-2d` to LARGE_DATASET.

3. Interference analysis

In the interference analysis, we measure the execution time of a *task under test* running on one of the cores, with the *interfering tasks* executing on the remaining cores. The *task under test* is either one of our synthetic benchmarks or one of the Polybench benchmarks (see Section 2), while the *interfering tasks* are one of our synthetic benchmarks. We chose to include the Polybench as *task under test* in our interference analysis to better represent how real-life workloads may be subject to interference on our target platforms (*Xilinx ZU9EG* and *Nvidia TX2*). *Interfering tasks* access memory at a controllable intensity by changing the ratio between $Mops$ and $Cops$ (see Section 2). By tuning this ratio, we regulate the percentage of total memory bandwidth that *interfering tasks* are allowed to consume. We report this percentage in the plot on the X axis as throttling parameter *THR%*.

Figs. 4(a), 4(b), 4(c) and 4(d), 4(e), 4(f) present the interference analysis results obtained using the *Nvidia TX2* and the *Xilinx ZU9EG*, respectively. There is one subplot for each interference traffic type (*READ_MISS*, *WRITE_MISS*, *MEMSET*). The experiments were run 10 times for each benchmark due to the high amount of time some of the Polybench require to run, especially when subject to interference. The results we obtained had really low standard deviation when compared to the mean (since all the results were within 5% of the average), as such, the average of the measured slowdown was plotted for all the experiments in Fig. 4. The slowdown curve of the synthetic benchmarks running as *task under test* are shown as black curves, whereas the slowdown curve of the 30 Polybench benchmarks are presented in the form of areas (i.e., grouped), with a label indicating how many Polybench benchmarks fall in one specific category. The perimeter of the areas is obtained by merging the highest and lowest values of all the slowdown curves of the Polybench which are part a specific category as the *THR%* changes (with the special case of the area becoming a line when only one benchmark fits in a category). This process is shown with an example in Fig. 3. The categories in Fig. 4 are: (i) **ABOVE**: those benchmarks that suffer more interference than the *READ_MISS* synthetic benchmark throughout the entire *THR%* range; (ii) **CROSSING**: those benchmarks that suffer more interference than the *READ_MISS* synthetic benchmark only for some *THR%* configurations; (iii) **BELOW**: those benchmarks that suffer less interference than the *READ_MISS* synthetic benchmark. *READ_MISS* (black curve with blue circular points) was used as the divider between categories as it was assumed by previous works [5,19–21] to be the *task under test* most sensitive to memory interference. If that assumption were true, no benchmarks should fall into the **ABOVE** or **CROSSING** categories in any part of Fig. 4. However, one can quickly notice that is not the case. In general, it can be observed that the *READ_MISS* synthetic benchmark is not the task most subject to interference in any of the subplots. Not only that, but in all subplots some of the Polybench are more subject to interference than any of the synthetic benchmarks.

As an example, subplots 4(a) and 4(d) present the slowdown perceived by real and synthetic benchmarks when running in the presence of *READ_MISS interfering tasks*, an interference configuration which some literature [5,20,21] considered to be the worst-case one. Subplot 4(a) shows that, for the *Nvidia TX2* platform, two Polybench fall into the **ABOVE** category, while twelve belong to the **CROSSING** category. On the other hand, subplot 4(d) shows that, for the *Xilinx ZU9EG* platform, four Polybench fall within the **ABOVE** category, and two belong to the **CROSSING** one. Note that these results are not captured by the assumption made in the previously cited works.

Subplots 4(b) and 4(c) show that the worst-case interference on *Nvidia TX2* is generated by *WRITE_MISS* (Subplot 4(b)), which causes R/W traffic due to the cache-line update operation. This result is in line with how other literature [6,10,19] conducts interference analysis. However, subplots 4(e) and 4(f) show that on the *Xilinx ZU9EG* hardware the worst-case interference traffic type is *MEMSET* (Subplot 4(f)), which causes slowdowns of up to ≈12×. This is a type of interference which is not often investigated, even among literature which is more thorough and checks for *WRITE_MISS* traffic.

3.1. Analysis summary

The results presented in this Section demonstrate that on the *Xilinx ZU9EG* and the *Nvidia TX2*, *READ_MISS* is neither the workload causing the highest amount of interference (highlighted by subplots 4(b) and 4(f)) nor the one most sensitive to memory interference (as seen in all subplots of Fig. 4). In fact, the results are highly platform-dependent, with the two analyzed MSoCs having different worst-case *interfering tasks* (*WRITE_MISS* on the *Nvidia TX2* and *MEMSET* on the *Xilinx ZU9EG*). Fig. 4, in general, highlights the need to do proper memory interference characterization, to determine the actual worst-case: it is not possible to make generalizations. Finally, the results also show that some Polybench are more subject to interference than any of the synthetic benchmarks. As the synthetic benchmarks are also the *interfering tasks*, and they interact with the DRAM due to the way they were constructed, this may seem counterintuitive.

In the next section, we explore why specific real-world benchmarks are more sensitive to memory interference than synthetic ones.

4. Last level cache thrashing

Further investigation is required to understand why some real-world workloads suffer from more interference than synthetic ones — which are modeled to capture the worst-case behaviors. When conducting interference analysis, it can be misleading to exclusively focus on DRAM interference. Cache events can also play a pivotal role in execution time increase. In this section, we thoroughly analyze and quantify the effects of cache interference, providing a correlation between the traffic

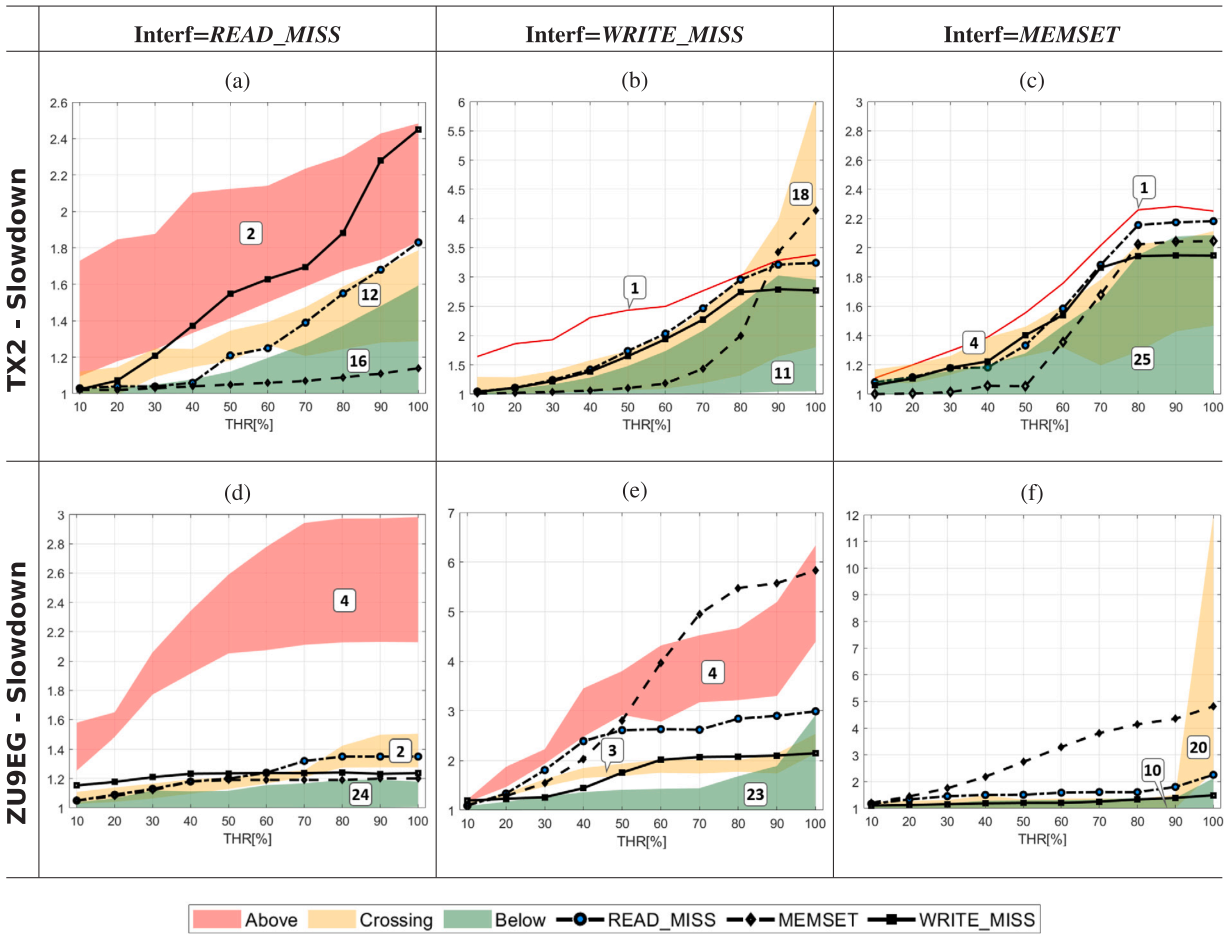

Fig. 4. ***Nvidia TX2*** **(a,b,c),** ***Xilinx ZU9EG*** **(d,e,f).** Time increase for Synthetic and Polybench benchmarks. Note: each plot has a different maximum y axis value.

generated by the *interfering tasks*, and their effect on the memory performance of our evaluation setup. Cache interference has been addressed in the literature using techniques like *shared cache partitioning* [27,28]. In this section, we also expose a micro-architectural cache interference effect reported on the *Xilinx ZU9EG* that may not be solved through the previously mentioned techniques.

For this evaluation, we profile the execution of the *READ_MISS* synthetic benchmark acting as a *task under test*, under different interference configurations. We focus on this benchmark because it is the only one that generates regular cache memory traffic. Therefore, it can be easily tracked and analyzed. The other benchmarks (i.e., *WRITE_MISS* and *MEMSET*), generate a memory traffic that either pollutes the cache (i.e., *WRITE_MISS*), making it difficult to analyze the cache effects, or does not access the cache memory at all (i.e., *MEMSET*, see Section 2). During each test run, we keep track of the LLC refills using Performance Monitoring Units (PMUs) [29]. Since the two hardware platforms that we use have different memory configurations (i.e., 1 MB and 2 MB LLC cache size for the *Xilinx ZU9EG* and the *Nvidia TX2*, respectively), we use two different sets of memory footprint (i.e., *fp*) for the *task under test* in the two setups. For the *Xilinx ZU9EG*, we use buffer sizes of 512 KB, 768 KB, 2048 KB, and 16384 KB. For the *Nvidia TX2*, we use 1024 KB, 1536 KB, 4096 KB, and 32768 KB buffers. Overall, the idea is to use memory footprints that are respectively smaller and larger than the LLC size in both setups, to emphasize the effects of the LLC cache behavior in the results. The interference is generated using our synthetic benchmarks – one for each remaining core – configured with a fixed *fp* of 16 MB, which is way larger than the LLC size of both platforms. This ensures that the *interfering tasks* cause interference at each level of the memory hierarchy. For each type of memory interference, we repeat the execution of the experiment, with varying values of the *THR%* parameter (see Section 3) of the *interfering tasks*. These configurations of the synthetic benchmarks require a really low amount of time to run. As such, the results reported in this section are the average of the measurements taken by doing 1000 runs for each value of *THR%* (in general, the results were within 5% of the average).

Figs. 5 and 6 present the results of the experiments for the *Nvidia TX2* and the *Xilinx ZU9EG*, respectively. In particular, Figs. 5(a), 5(b), 5(c) and 6(a), 6(b), 6(c) present the **slowdown** of the *task under test*, while 5(d), 5(e), 5(f) and 6(d), 6(e), 6(f) present the **LLC refill** results. In the figures, each subplot presents the results as a function of both the type of interference generated by the *interfering tasks* (i.e., *READ_MISS*, *WRITE_MISS* and *MEMSET*) and the *THR%* parameter configuration. In the following sections, we discuss the results for each configuration of the *interfering tasks*.

4.1. *READ_MISS interfered by READ_MISS*

Subplots 5(a), 5(d), and 6(a), 6(d) show the results obtained by using *READ_MISS* as *interfering tasks*. As expected, the results heavily depend on the *fp* configuration of the *task under test*. If the *fp* of the *task under test* is larger than the LLC size, each *load* operation performed triggers an LLC refill, regardless of the *THR%* setting used for the *interfering tasks*. This is highlighted in Fig. 5(d) for *fp* configurations of 4096K and 32768K, and Fig. 6(d) for configurations of 2048K and 16384K. As a result, the number of LLC-to-DRAM transactions is constant, and the slowdown shown in Figs. 5(a) and 6(a) is entirely caused by DRAM interference. We notice that this phenomenon leads to a 3× slowdown on the *Nvidia TX2* board and to a 1.5× slowdown on

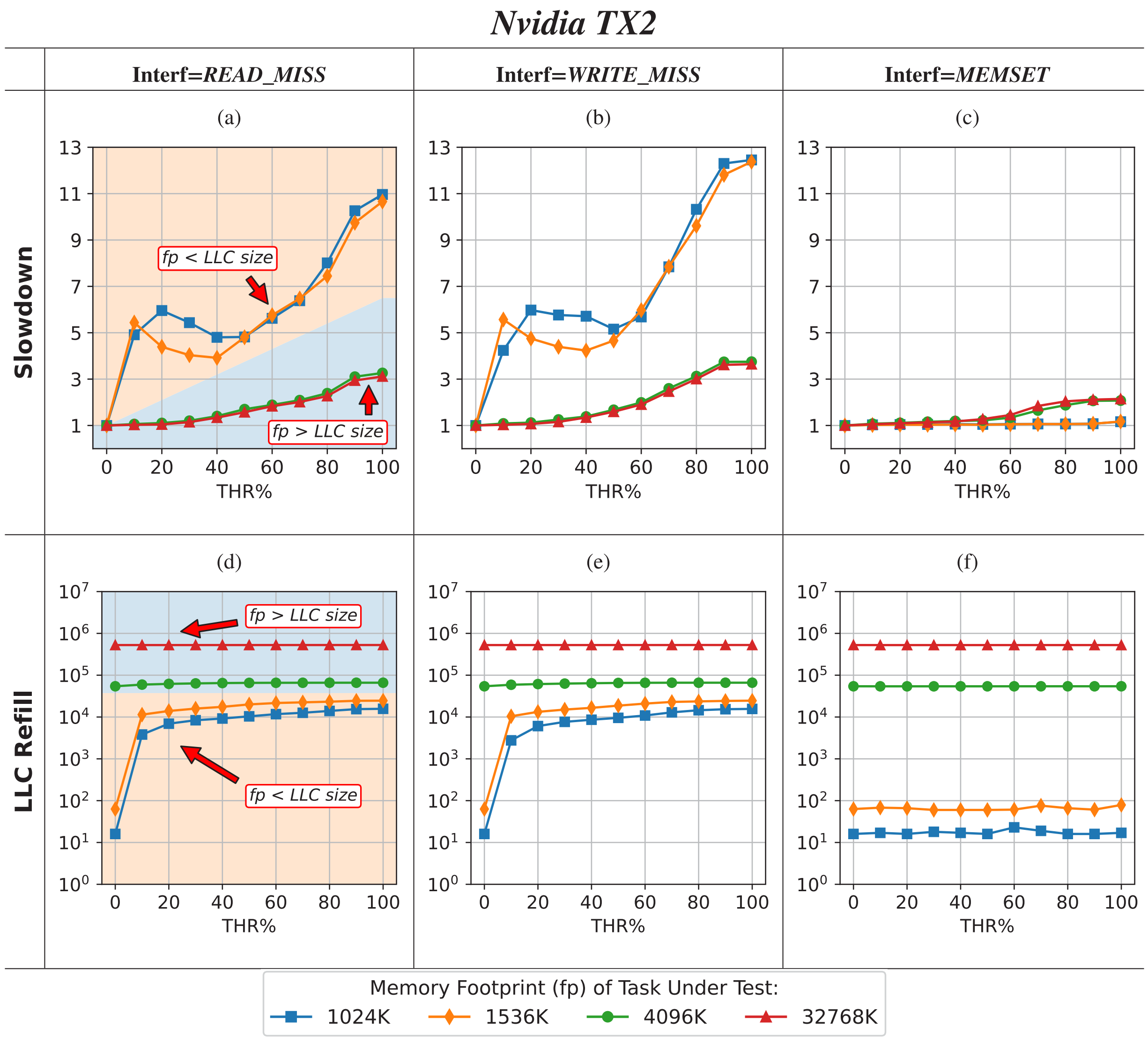

Fig. 5. ***Nvidia TX2.*** Slowdown and LLC refills triggered by the *task under test* as a function of the *interfering tasks* configuration.

the *Xilinx ZU9EG*. On the other hand, when fp is smaller than the LLC size and the *interfering tasks* are muted (i.e., 0% *THR%*), the memory buffer used by the *task under test* is entirely cached, and the benchmark generates a negligible amount of LLC refills. This is reported in Fig. 5(a) for fp configurations of 1024K and 1536K, and in Fig. 6(a) for configurations of 512K and 768K. As the *THR%* grows, we observe an increasing number of LLC refills due to the cache space contention generated by the *interfering tasks*. In this case, the slowdown is caused by both the DRAM interference and the LLC eviction, and therefore, it is much more severe. Ultimately, the results report a slowdown factor of 11× on the *Nvidia TX2* platform (Fig. 5(a)), and 4.5× on the *Xilinx ZU9EG* (Fig. 6(a)).

4.2. *READ_MISS interfered by WRITE_MISS*

Figs. 5(b), 5(e), 6(b), and 6(e) report the slowdown and LLC refill results obtained using *WRITE_MISS* as *interfering tasks*. In this configuration, the results reported in Figs. 5(e) and 6(e) show that the LLC refills are comparable to the ones registered in the *READ_MISS interfered by READ_MISS* configuration (Figs. 5(d) and 6(d)). However, the slowdown results (Figs. 5(b), 6(b)) are sensibly higher, especially on the *Xilinx ZU9EG* platform. To explain such a difference we repeat the benchmark execution using PMUs to profile the number of LLC writebacks performed by the *task under test*.

The LLC writeback results are reported in Figs. 7 and 8. In case of *READ_MISS* interference (Figs. 7(a), 8(a)), the number of LLC writebacks is less than 10^3 for each fp configuration that is used. In the same configuration, the number of LLC refills (see Figs. 5(d), 6(d)) reaches $10^5 - 10^6$. Therefore, the number of LLC writebacks represents less than 1% of the total number of LLC-to-DRAM transactions, and can be considered as negligible. This is because the traffic generated by the *task under test* and the *interfering tasks* is read-only, and no cache line is written back to DRAM memory. On the other hand, the number of LLC writebacks registered in the *READ_MISS interfered by WRITE_MISS* configuration (Figs. 7(b), 8(b)) grows accordingly to the *THR%* parameter of the *interfering tasks*. This happens because both platforms implement a writeback caching policy, therefore, dirty cached data is evicted to memory only when the CPU cores try to access the corresponding cache line, as explained in Section 2. The *task under test*, generating read-only traffic, triggers a number of LLC refills that evict (i.e., write back to DRAM) the cache lines dirtied by the *interfering tasks*. Ultimately, this leads to an increase in the number of LLC writebacks transactions over the *worst-case* registered by the *READ_MISS interfered by READ_MISS* configuration, and therefore to a higher slowdown in the benchmark execution time. This configuration represents the worst-case slowdown scenario for the *Nvidia TX2* platform, as the reported slowdown reaches up to a factor 12.5×. Whereas, on the *Xilinx ZU9EG*, we register a 10× increase in execution time.

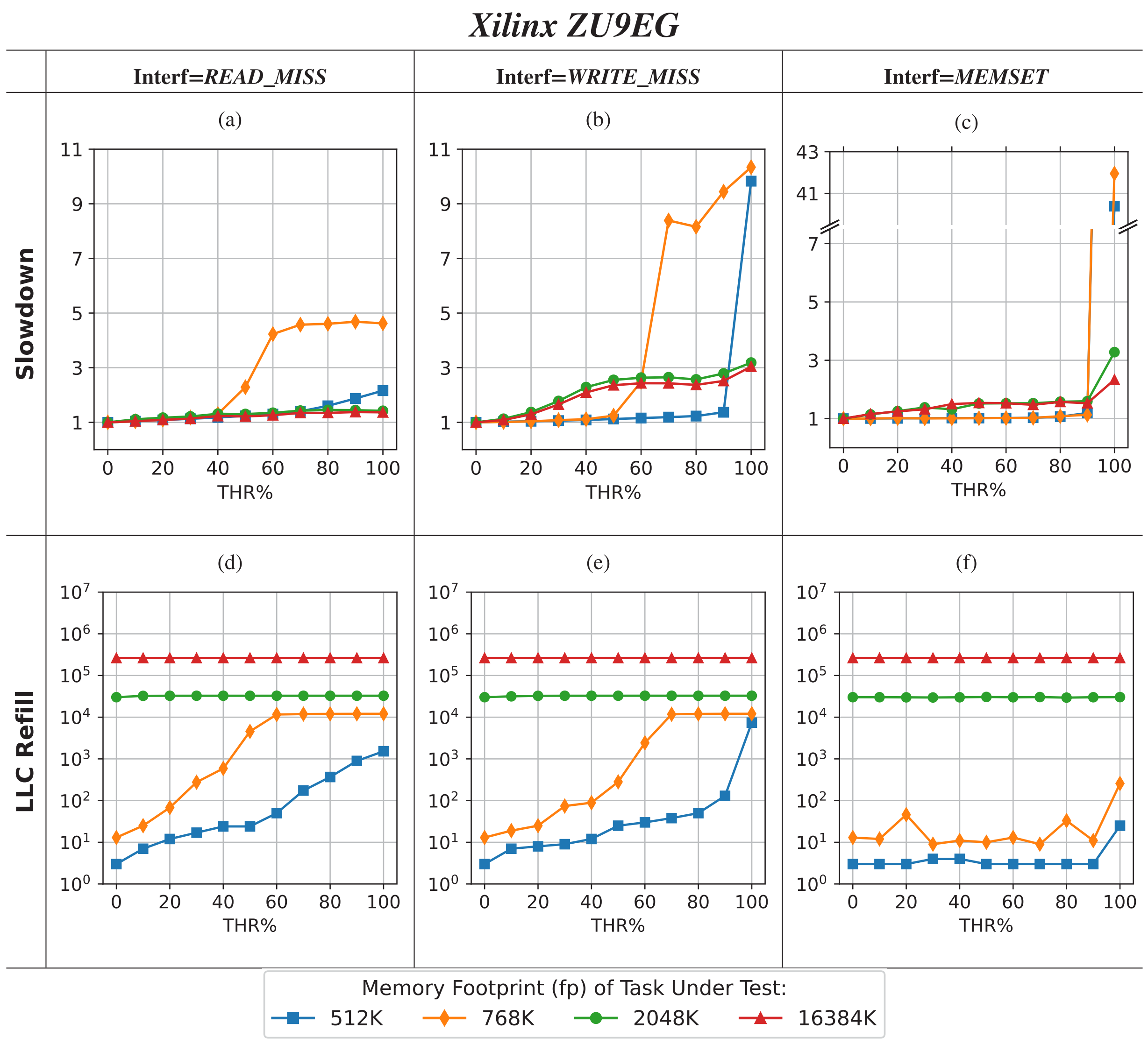

Fig. 6. ***Xilinx ZU9EG.*** Slowdown and LLC refills triggered by the *task under test* as a function of the *interfering tasks* configuration.

4.3. READ_MISS interfered by MEMSET

As explained in Section 2, the traffic generated by *MEMSET interfering tasks* is 100% *write-no-allocate*. This write-only traffic directly writes out to DRAM without causing any cache LLC refills/writebacks. The results obtained by executing the *task under test* with this type of *interfering tasks*, reported in Figs. 5(c), 5(f), 7(c), and 6(c), 6(f), 8(c) demonstrate this aspect. In fact, the impact of the *THR%* parameter on the number of LLC refills/writebacks is almost null or negligible if compared to other configurations (see Sections Section 4.1, 4.2). Nevertheless, the slowdown results are very different between the two setups. On the *Nvidia TX2* (Fig. 5(c)), the results register a slowdown factor of 2.5× when the *fp* is larger than the LLC size, whereas no slowdown can be noticed if the *fp* is smaller than the LLC size. Note that this is the expected result since the *interfering tasks* do not affect the behavior of the cache and therefore do not create cache space contention. On the *Xilinx ZU9EG* (Fig. 6(c)), we obtain a 2.5×-3.5× slowdown when the *fp* is larger than the LLC size, which is comparable to the one obtained using *WRITE_MISS* interference, and an unexpected **42**× slowdown when the *fp* is smaller than the LLC size. To understand the cause of this slowdown, we better analyze the memory architecture of the *Xilinx ZU9EG*.

The *Xilinx ZU9EG* implements a non-blocking cache: a cache memory that can simultaneously handle CPU requests and linefills/writebacks [30]. In this cache, a store buffer temporarily holds writeback operations that exit the cache memory (evictions). This way, the cache can continue serving CPU requests while completing writebacks. On the *Xilinx ZU9EG*, the store buffer is also used to hold and merge *write-no-allocate* transactions. By merging multiple transactions into a single memory burst, the *write-no-allocate* traffic suffers less per-transaction overhead, enjoying a higher memory bandwidth [30]. However, the store buffer has a limit: if it becomes full (e.g., saturated by requests), the cache blocks and no longer accepts CPU requests [31]. This causes the CPU cores to stall until the entries in the store buffer are freed (written back to memory).

Cache blocking can happen in the *READ_MISS* interfered by *MEMSET* scenario: the *write-no-allocate* traffic generated by the *MEMSET interfering tasks* can saturate the store buffer, causing the LLC cache to block. This, in turn, can cause the CPU core executing the *READ_MISS* task to stall. To demonstrate this phenomenon, we repeated the execution of the *task under test* (fp = 512 KB) both in isolation and co-scheduled with *MEMSET interfering tasks*. During the execution, we used PMUs to profile the execution of both the *task under test* and the *interfering tasks*. For the *task under test*, we configured PMUs to track the *CPU stalls because of load misses* event (PMU event number

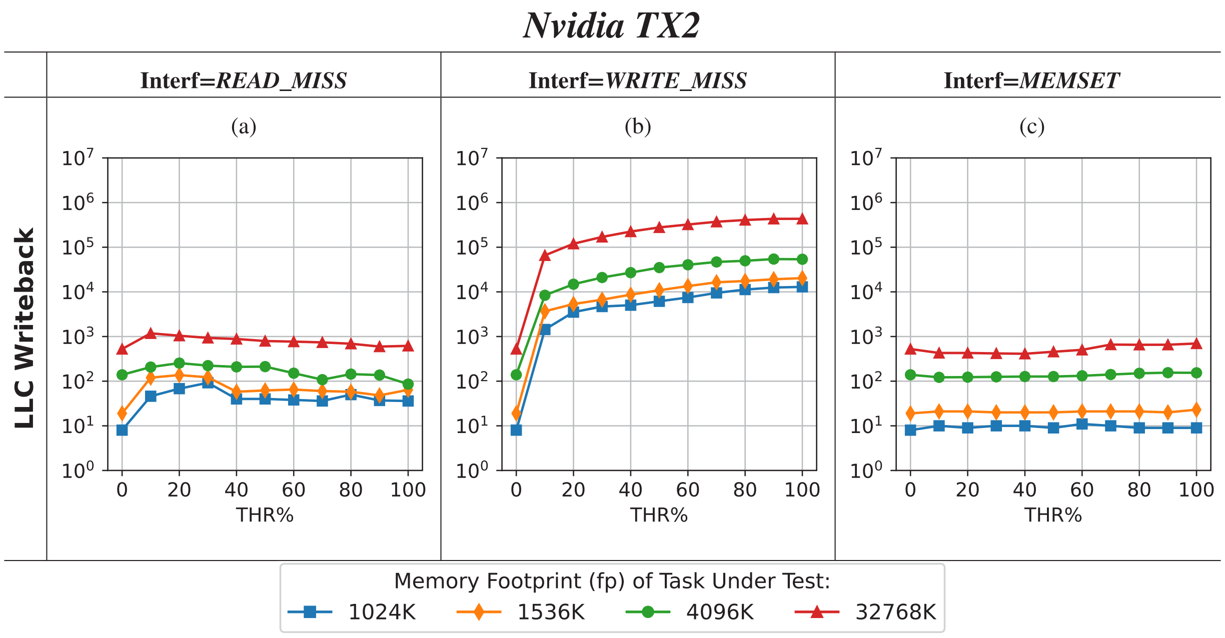

Fig. 7. ***Nvidia TX2.*** LLC writebacks triggered by the *task under test* as a function of the *interfering tasks* configuration.

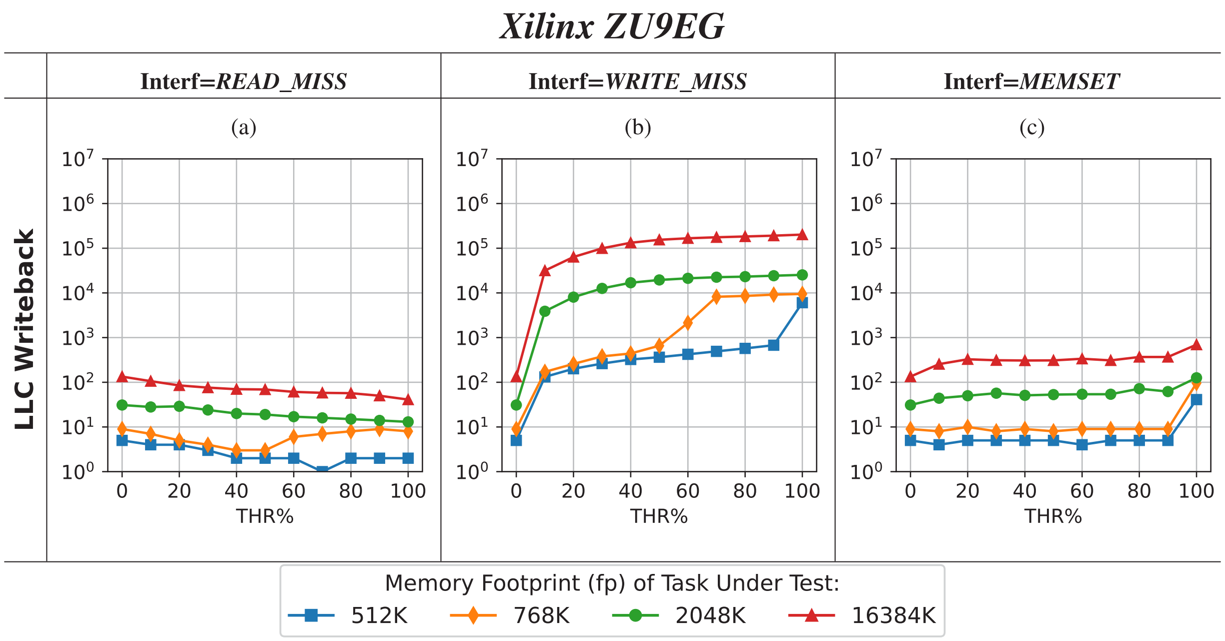

Fig. 8. ***Xilinx ZU9EG.*** LLC writebacks triggered by the *task under test* as a function of the *interfering tasks* configuration.

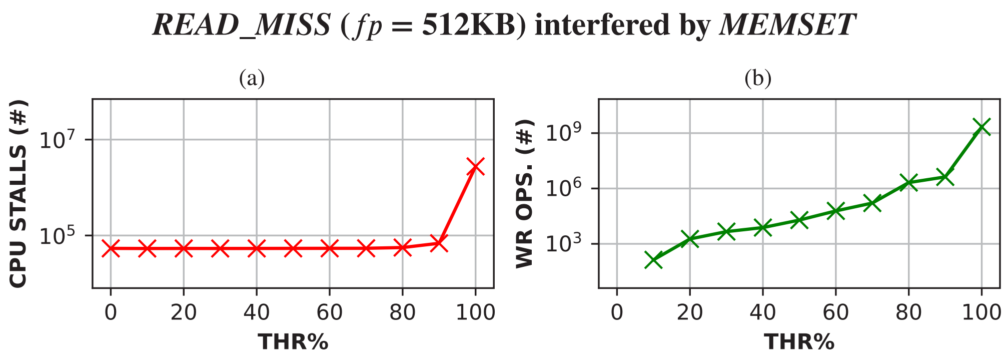

Fig. 9. *READ_MISS* (fp = 512 KB) interfered by *MEMSET*. The *CPU STALLS because of load miss* (Subplot 9(a)) are measured on the *task under test*. The *WRITE OPERATIONS that stall the pipeline because the store buffer is full* (Subplot 9(b)) are measured on the *interfering tasks*.

0xE7). For the *interfering tasks*, we configured the PMUs to track the *WRITE OPERATIONS that stall the pipeline because the store buffer is full* event (PMU event number 0xC7). The results are reported in Fig. 9. In the figure, the subplots present the results as a function of the *THR%* parameter of the *interfering tasks*. As shown in Subplot 9(b), when the *THR%* parameter changes from 90% to 100%, the number of *WRITE OPERATIONS that stall the pipeline because the store buffer is full* grows from $\approx 10^6$ to $\approx 10^9$. Accordingly, the number of *CPU stalls because of load misses* (Subplot 9(a)) grows from $\approx 10^5$ to $\approx 10^7$. This result indicates that when the *THR%* parameter reaches 100%, the store buffer gets saturated by the memory requests of the *MEMSET interfering tasks*, causing the CPU core executing the *task under test* to stall, and ultimately, the observed slowdown.

4.4. Analysis summary

This analysis allows us to draw several results: (i) First, we observe that cache-bound benchmarks (i.e., fp < LLC size) suffer from greater slowdown when compared to the DRAM-bound ones (see Section 4.1). This phenomenon demonstrates that, on our platforms, the cache contention problem is more severe than DRAM interference. (ii) Second, we observe that the performance of the *task under test* is not only influenced by DRAM interference and cache space contention, but also by additional interference from cache maintenance operations (i.e., LLC writebacks, see Section 4.2). This has proven to be a major source of slowdown on our setup, particularly on *Xilinx ZU9EG*. (iii) Lastly, we note that the combination of different memory access patterns can generate a slowdown which is not exclusively caused by interference but also by architectural phenomena, which act as hardware bottlenecks in the memory hierarchy (see Section 4.3).

5. Related work

Many papers address the problem of memory interference in the context of multicore SoCs [1,2,4–8,21,25,32–34]. Typically, they fall into one of three main categories: platform-specific memory characterization, DRAM maximum interference estimate and mitigation, and cache access predictability.

Platform-specific memory characterization. The problem of identifying interference effects in modern MSoCs is very relevant, and many works have analyzed the interference characteristics on different types of COTS MSoCs (mainly FPGA-based SoC, GP-GPU-based SoC). Bansal et al. [25] propose a deep characterization of the memory systems present on the ZU9EG, with some focus on the interference which the CPU cores can cause to each other. Manev et al. [32] tried to analyze the memory performance and requirements of accelerators on ZU9EG and ZU3EG. Capodieci et al. [21] highlight how on *Nvidia*'s platforms, instead, DRAM performance has evolved over the years, with the rapid increase in GPU performance.

In this paper, we also performed extensive tests on different embedded multicore SoCs from different vendors and with different features to characterize various possible interference effects.

DRAM maximum interference estimate and mitigation. Memory interference in MSoC has been a significant research focus since these systems were introduced. Several studies have concentrated on estimating the Worst-Case Execution Time (WCET) using various analytical methods. Andreozzi et al. [12] employed mixed integer linear programming (MILP) to provide a rigorous estimation. Other works have also utilized synthetic loads. Radojkovi et al. [5] focused on singular load types. Nowotsch et al. [6] instead combined different types of memory access patterns, such as *WRITE_MISS* and *READ_MISS*. Hyoseung et al. [11] instead aimed at defining boundaries for memory-based interference from CPU cores within MSoC. In addressing the mitigation and management of memory access and interference, various research efforts have explored scheduling mechanisms to reduce conflicts when multiple cores access shared hardware resources such as DRAM [13,14,16,19,35]. Together, these approaches represent a range of strategies for reducing memory contention and ensuring more predictable performance in MSoC.

Cache access predictability. In addition to managing memory access and interference, significant research has focused on the control and management of shared cache interference in MSoC systems. Dugo et al. [36] leverage cache locking and partitioning to reduce non-determinism and contention in lower-level caches, which improves timing performance. Kaushik et al. [37,38] address cache coherence through specialized protocols. They propose a predictable modify-share-invalid (MSI) protocol and a modify-exclusive-share-invalid (MESI) protocol, which use specific design invariants to ensure predictability and minimize cache interference.

Other researchers focus on analyzing cache usage fairness. Seongbeom et al. [39] present a scheme which demonstrates that fair cache usage directly correlates with improved execution time, providing a different perspective on managing cache resources effectively.

Valsan et al. [8] present a different cache mitigation technique, tested on COTS multicore systems, which provides better guarantees than the classic cache partitioning for their tested workloads.

In summary, while various strategies exist to handle shared cache interference, their effectiveness can vary significantly based on the architecture and workload. As demonstrated by the characterization in this paper, careful attention must be paid to choosing the proper techniques to ensure predictable and efficient performance in multicore SoC environments.

Proper worst-case interference estimate. There are some examples of underestimating the worst-case in the literature which we already cited.

Capodieci et al. [21] measure the effect of memory interference on *Nvidia*'s platforms (including on *Nvidia TX2*). In their experiments they use *READ_MISS* to measure interference effects as the *task under test*, which was proven as the task not most subject to interference in Section 3. This means that the worst-case when it comes to DRAM-based interference is underestimated. Cavicchioli et al. [20] present the technique of Controlled Memory Request Injection (CMRI) and evaluate its effectiveness with synthetic benchmarks run on the *Nvidia TX2*. However, they try to model worst-case interference by using *READ_MISS* as both the *task under test* and the *interfering tasks*, which means that the worst-case is underestimated, as the results in Section 3 highlight. Brilli et al. [19] present a work on modern HeSoC memory regulation and control, which expands on the work from Cavicchioli et al. [20]. They evaluate CMRI on both the *Nvidia TX2* and *Xilinx ZU9EG*. The previous analysis is expanded with *WRITE_MISS* as another possible *interfering task*. However, in their experiments they: (i) do not take into consideration *MEMSET* as a *interfering task* (important for the *Xilinx ZU9EG*); (ii) still only consider *READ_MISS* as the most interfered *task under test*. This means that they also underestimated the worst-case interference with which to compare their mitigation technique. Hyoseung et al. [10] try to bound memory interference delay in COTS MSoCs. During the evaluation, they make use of the *STREAM* benchmark [40] as the *interfering task*. The *STREAM* benchmark at its highest level of memory intensity acts like a memcpy, which has a memory access pattern very similar to *WRITE_MISS*. While this is fine for the case examined in the paper, on a platform like the *Xilinx ZU9EG* using this benchmark would not expose the interference effects which happen with *MEMSET* as the *interfering task*. Nowotsch et al. [6] in their maximum DRAM interference estimation only investigate the interference of *WRITE_MISS* and *READ_MISS* traffic, meaning that their methodology may not find the actual worst-case on some platforms, as we have demonstrated in this paper. Radojkovi et al. [5] present WCET bound estimates which also make use of *READ_MISS* as both the *task under test* and the interfering task, without proper checks for other traffic types. This leads to the possibility of a WCET bound estimate significantly lower than the real WCET, as we observed for both the *Nvidia TX2* and the *Xilinx ZU9EG*.

6. Discussion and conclusion

In this work, we presented a thorough analysis of DRAM and intra-cluster interference effects on two representative multicore SoCs: the *Nvidia TX2* and the *Xilinx ZU9EG*. We show that DRAM-bound read-only and write-only traffics generated by synthetic workloads do not represent worst-case in terms of sensitivity to memory interference, that different workloads generate different levels of interference and that there is no single traffic type which generates the highest amount of interference on all platforms. To find the worst-case for the hardware under analysis, an in-depth interference characterization is necessary. We highlighted that the impact of interference depends on the memory hierarchy level where it occurs: we observed up to 3× slowdown due to DRAM-interference, up to 13× slowdown due to the combination of LLC space contention and DRAM interference, and a 42× slowdown due to an architectural bottleneck in the memory hierarchy of the *Xilinx ZU9EG*. With these high slowdowns we present, we explain in this paper why, counterintuitively, some tasks which do not make particular use of the main memory under normal circumstances can be subject to more slowdown than synthetic benchmarks when subject to intra-cluster interference. There are multiple ways in which current and future works may be affected by our findings and from the methodology we used in this paper.

Memory interference mitigation techniques [14,16,41,42] may be able to make use of the characterization and findings introduced in this paper to create a more precise interference analysis for their hardware, more aware of the limitations of their platforms. This improved analysis could then be used to better estimate the interference that a platform is subject to at runtime. With a more precise interference estimate, these mitigation techniques would then be able to more precisely limit the bandwidth of non-critical tasks, instead of taking conservative approaches which exclusively prioritize the timing guarantees of time critical tasks. This would in turn reduce bandwidth regulation inefficiency, leading to reduced waste of hardware resources [20].

Similar considerations apply to those approaches that rely on WCET estimation to derive formal schedulability analysis [12,43,44]. The more thorough interference characterization could lead to better WCET boundaries in these papers, leading to less conservative results. This applies to approaches based on both formal analysis [44] and heuristics [12].

The methodology applied in this work can help study interference caused by intra-cluster and inter-cluster phenomena. Systematic approaches to perform memory interference characterization [45] can build upon the characterization presented in this paper to evaluate a wider spectrum of architectural phenomena than they already do.

Finally, our own ongoing work is built upon this methodology: by first creating a thorough interference characterization of the platform under consideration using the approach presented in this paper, it is possible to estimate the level of bandwidth regulation needed to reduce the slowdown of a critical task exactly to a precise quality of service (QoS) value, with very low error. This allows quick and precise bandwidth regulation, which in turn leads to proper throttling of non-critical tasks. Thanks to this approach based on the interference characterization, we observe very low hardware under-utilization in our experiments, while still reducing the slowdown to the expected QoS value.

CRediT authorship contribution statement

Lorenzo Carletti: Writing – original draft, Visualization, Validation, Software, Investigation, Formal analysis, Data curation. **Andrea Serafini:** Writing – original draft, Visualization, Validation, Software, Investigation, Formal analysis, Data curation. **Gianluca Brilli:** Writing – original draft, Visualization, Validation, Software, Resources, Investigation, Formal analysis, Data curation. **Alessandro Capotondi:** Writing – review & editing, Supervision, Resources. **Alessandro Biasci:** Writing – review & editing, Supervision, Resources, Project administration. **Paolo Valente:** Writing – review & editing, Supervision, Resources, Methodology, Conceptualization. **Andrea Marongiu:** Writing – review & editing, Supervision, Resources, Project administration, Methodology, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] M. Mattheeuws, B. Forsberg, A. Kurth, L. Benini, Analyzing memory interference of FPGA accelerators on multicore hosts in heterogeneous reconfigurable SoCs, in: Design, Automation & Test in Europe Conference & Exhibition, DATE, 2021, pp. 1152–1155, http://dx.doi.org/10.23919/DATE51398.2021.9473925.

[2] G. Brilli, A. Capotondi, P. Burgio, A. Marongiu, Understanding and mitigating memory interference in FPGA-based hesocs, in: Design, Automation & Test in Europe Conference & Exhibition, DATE, 2022, pp. 1335–1340, http://dx.doi.org/10.23919/DATE54114.2022.9774768.

[3] Y.J. Lin, C.L. Yang, J.W. Huang, T.J. Lin, C.W. Hsueh, N. Chang, System-level performance and power optimization for MPSoC: A memory access-aware approach, ACM Trans. Embed. Comput. Syst. 14 (1) (2015) http://dx.doi.org/10.1145/2656339.

[4] J. Boudjadar, J.H. Kim, S. Nadjm-Tehrani, Performance-aware scheduling of multicore time-critical systems, in: ACM/IEEE International Conference on Formal Methods and Models for System Design, 2016, pp. 105–114, http://dx.doi.org/10.1109/MEMCOD.2016.7797753.

[5] P. Radojkovic, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, F. Cazorla, On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments, TACO 8 (2012) 34, http://dx.doi.org/10.1145/2086696.2086713.

[6] J. Nowotsch, M. Paulitsch, Leveraging multi-core computing architectures in avionics, in: Ninth European Dependable Computing Conference, 2012, pp. 132–143, http://dx.doi.org/10.1109/EDCC.2012.27.

[7] M. Bechtel, H. Yun, Memory-aware denial-of-service attacks on shared cache in multicore real-time systems, IEEE Trans. Comput. 71 (9) (2022) 2351–2357, http://dx.doi.org/10.1109/TC.2021.3108044.

[8] P.K. Valsan, H. Yun, F. Farshchi, Taming non-blocking caches to improve isolation in multicore real-time systems, in: IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2016, pp. 1–12, http://dx.doi.org/10.1109/RTAS.2016.7461361.

[9] T. Kloda, G. Gracioli, R. Tabish, R. Mirosanlou, R. Mancuso, R. Pellizzoni, M. Caccamo, Lazy load scheduling for mixed-criticality applications in heterogeneous MPSoCs, ACM Trans. Embed. Comput. Syst. 22 (3) (2023) http://dx.doi.org/10.1145/3587694.

[10] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, R. Rajkumar, Bounding memory interference delay in COTS-based multi-core systems, in: 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium, RTAS, 2014, pp. 145–154, http://dx.doi.org/10.1109/RTAS.2014.6925998.

[11] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, R. Rajkumar, Bounding and reducing memory interference in COTS-based multi-core systems, Real-Time Syst. 52 (3) (2016) 356–395, http://dx.doi.org/10.1007/s11241-016-9248-1.

[12] M. Andreozzi, A. Frangioni, L. Galli, G. Stea, R. Zippo, A MILP approach to DRAM access worst-case analysis, Comput. Oper. Res. 143 (C) (2022) http://dx.doi.org/10.1016/j.cor.2022.105774.

[13] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, R. Kegley, A predictable execution model for COTS-based embedded systems, in: 17th IEEE Real-Time and Embedded Technology and Applications Symposium, 2011, pp. 269–279, http://dx.doi.org/10.1109/RTAS.2011.33.

[14] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, L. Sha, MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms, in: IEEE 19th Real-Time and Embedded Technology and Applications Symposium, RTAS, 2013, pp. 55–64, http://dx.doi.org/10.1109/RTAS.2013.6531079.

[15] C. Courtaud, J. Sopena, G. Muller, D. Gracia Pérez, Improving prediction accuracy of memory interferences for multicore platforms, in: IEEE Real-Time Systems Symposium, RTSS, 2019, pp. 246–259, http://dx.doi.org/10.1109/RTSS46320.2019.00031.

[16] H. Yun, W. Ali, S. Gondi, S. Biswas, BWLOCK: A dynamic memory access control framework for soft real-time applications on multicore platforms, IEEE Trans. Comput. 66 (7) (2017) 1247–1252, http://dx.doi.org/10.1109/TC.2016.2640961.

[17] A. Ahmed, K. Skadron, Hopscotch: a micro-benchmark suite for memory performance evaluation, in: Proceedings of the International Symposium on Memory Systems, MEMSYS '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 167–172, http://dx.doi.org/10.1145/3357526.3357574.

[18] J. McCalpin, Memory bandwidth and machine balance in high performance computers, in: IEEE Technical Committee on Computer Architecture Newsletter, 1995, pp. 19–25.

[19] G. Brilli, R. Cavicchioli, M. Solieri, P. Valente, A. Marongiu, Evaluating controlled memory request injection for efficient bandwidth utilization and predictable execution in heterogeneous socsf, ACM Trans. Embed. Comput. Syst. 22 (1) (2022) http://dx.doi.org/10.1145/3548773.

[20] R. Cavicchioli, N. Capodieci, M. Solieri, M. Bertogna, P. Valente, A. Marongiu, Evaluating controlled memory request injection to counter PREM memory underutilization, in: Job Scheduling Strategies for Parallel Processing, Springer International Publishing, Cham, 2020, pp. 85–105, http://dx.doi.org/10.1007/978-3-030-63171-0_5.

[21] N. Capodieci, R. Cavicchioli, I.S. Olmedo, M. Solieri, M. Bertogna, Contending memory in heterogeneous socs: Evolution in NVIDIA tegra embedded platforms, in: 2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA, 2020, pp. 1–10, http://dx.doi.org/10.1109/RTCSA50079.2020.9203722.

[22] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, J. Cavazos, Auto-tuning a high-level language targeted to GPU codes, in: Innovative Parallel Computing, InPar, 2012, pp. 1–10, http://dx.doi.org/10.1109/InPar.2012.6339595.

[23] ARM, Arm cortex-a53 mpcore processor technical reference manual r0p4, 2025, URL: https://developer.arm.com/documentation/ddi0500/j/ch06s02s05.

[24] NVIDIA, NVIDIA Jetson AGX Orin Series - Technical brief, 2021, URL: https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf.

[25] A. Bansal, R. Tabish, G. Gracioli, R. Mancuso, R. Pellizzoni, M. Caccamo, Evaluating the memory subsystem of a configurable heterogeneous MPSoC, in: Proceedings of the Operating Systems Platforms for Embedded Real-Time Applications, 2018.

[26] Carletti, Lorenzo and Brilli Gianluca, SynthMemBench, 2025, URL: https://github.com/LorenzoCarletti/SynthMemBench.

[27] N. Suzuki, H. Kim, D. de Niz, B. Andersson, L. Wrage, M. Klein, R. Rajkumar, Coordinated bank and cache coloring for temporal protection of memory accesses, in: IEEE 16th International Conference on Computational Science and Engineering, 2013, pp. 685–692, http://dx.doi.org/10.1109/CSE.2013.106.

[28] X. Zhang, S. Dwarkadas, K. Shen, Towards practical page coloring-based multicore cache management, in: Proceedings of the 4th ACM European Conference on Computer Systems, EuroSys '09, Association for Computing Machinery, New York, NY, USA, 2009, pp. 89–102, http://dx.doi.org/10.1145/1519065.1519076.

[29] J. Barrera, L. Kosmidis, H. Tabani, E. Mezzetti, J. Abella, M. Fernandez, G. Bernat, F.J. Cazorla, On the reliability of hardware event monitors in mpsocs for critical domains, in: Proceedings of the 35th Annual ACM Symposium on Applied Computing, SAC '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 580–589, http://dx.doi.org/10.1145/3341105.3373955.

[30] AMD, Zynq UltraScale+ device technical reference manual, 2015, URL: https://docs.amd.com/v/u/en-US/ug1085-zynq-ultrascale-trm.

[31] M. Bechtel, H. Yun, Denial-of-service attacks on shared cache in multicore: Analysis and prevention, in: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2019, pp. 357–367, http://dx.doi.org/10.1109/RTAS.2019.00037.

[32] K. Manev, A. Vaishnav, D. Koch, Unexpected diversity: Quantitative memory analysis for zynq UltraScale+ systems, in: International Conference on Field-Programmable Technology, ICFPT, 2019, pp. 179–187, http://dx.doi.org/10.1109/ICFPT47387.2019.00029.

[33] B. Forsberg, M. Solieri, M. Bertogna, L. Benini, A. Marongiu, The predictable execution model in practice: Compiling real applications for COTS hardware, ACM Trans. Embed. Comput. Syst. 20 (5) (2021) http://dx.doi.org/10.1145/3465370.

[34] M. Hassan, A. Kaushik, H. Patel, Exposing implementation details of embedded DRAM memory controllers through latency-based analysis, ACM Trans. Embed. Comput. Syst. 17 (5) (2018) http://dx.doi.org/10.1145/3274281.

[35] J.M. Aceituno, A. Guasque, P. Balbastre, J. Simó, A. Crespo, Hardware resources contention-aware scheduling of hard real-time multiprocessor systems, J. Syst. Archit. 118 (2021) http://dx.doi.org/10.1016/j.sysarc.2021.102223.

[36] A.T. Aurora Dugo, J.B. Lefoul, F.G. De Magalhaes, D. Assal, G. Nicolescu, Cache locking content selection algorithms for ARINC-653 compliant RTOS, ACM Trans. Embed. Comput. Syst. 18 (5s) (2019) http://dx.doi.org/10.1145/3358196.

[37] A.M. Kaushik, M. Hassan, H. Patel, Designing predictable cache coherence protocols for multi-core real-time systems, IEEE Trans. Comput. 70 (12) (2021) 2098–2111, http://dx.doi.org/10.1109/TC.2020.3037747.

[38] M. Hassan, A.M. Kaushik, H. Patel, Predictable cache coherence for multi-core real-time systems, in: Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2017, pp. 235–246, http://dx.doi.org/10.1109/RTAS.2017.13.

[39] S. Kim, D. Chandra, Y. Solihin, Fair cache sharing and partitioning in a chip multiprocessor architecture, in: 13th International Conference on Parallel Architecture and Compilation Techniques, 2004, pp. 111–122, http://dx.doi.org/10.1109/PACT.2004.1342546.

[40] J. McCalpin, Sustainable memory bandwidth in current high performance computers, 1995, URL: https://www.cs.virginia.edu/~mccalpin/papers/bandwidth/node2.html#SECTION00020000000000000000.

[41] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo, R. Mancuso, MemPol: Policing core memory bandwidth from outside of the cores, in: 2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium, RTAS, 2023, pp. 235–248, http://dx.doi.org/10.1109/RTAS58335.2023.00026.

[42] P. Sohal, R. Tabish, U. Drepper, R. Mancuso, E-WarP: A system-wide framework for memory bandwidth profiling and management, in: 2020 IEEE Real-Time Systems Symposium, RTSS, 2020, pp. 345–357, http://dx.doi.org/10.1109/RTSS49844.2020.00039.

[43] B. Andersson, H. Kim, D.D. Niz, M. Klein, R.R. Rajkumar, J. Lehoczky, Schedulability analysis of tasks with corunner-dependent execution times, ACM Trans. Embed. Comput. Syst. 17 (3) (2018) http://dx.doi.org/10.1145/3203407.

[44] A. Agrawal, R. Mancuso, R. Pellizzoni, G. Fohler, Analysis of dynamic memory bandwidth regulation in multi-core real-time systems, in: 2018 IEEE Real-Time Systems Symposium, RTSS, 2018, pp. 230–241, http://dx.doi.org/10.1109/RTSS.2018.00040.

[45] A. Stevanato, M. Zini, A. Biondi, B. Morelli, A. Biasci, Learning memory-contention timing models with automated platform profiling, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 43 (11) (2024) 3816–3827, http://dx.doi.org/10.1109/TCAD.2024.3449237.

Lorenzo Carletti is a Ph.D. student at the University of Modena and Reggio Emilia. His research interest is mainly focused on the analysis and software-based mitigation of memory interference effects in multicores systems, with special emphasis on FPGA-based heterogeneous systems.

Andrea Serafini is an industrial Ph.D. student at the University of Modena and Reggio Emilia and Evidence SRL (Pisa, Italy). He previously graduated in Computer Science at the Physics, Informatics, and Mathematics Department at the University of Modena and Reggio Emilia. His research interest is mainly focused on the analysis and mitigation of memory interference effects in multicores and heterogeneous systems.

Gianluca Brilli is a postdoctoral researcher in computer engineering at the University of Modena and Reggio Emilia, within the High-Performance Real-Time Laboratory (HiPeRT-Lab) located in Modena, Italy. His expertise lies in the field of software and hardware acceleration using reconfigurable embedded systems. His main research interests are main memory QoS regulation and memory interference mitigation on FPGA-based heterogeneous systems.

Alessandro Capotondi (Member, IEEE) is Assistant Professor at the University of Modena and Reggio Emilia. He received a Ph.D. in Electronics, Telecommunications and Information Technology at the University of Bologna. He has been a research assistant and postdoctoral researcher at the University of Bologna and the University of Modena and Reggio Emilia. His research interests focus on embedded systems, heterogeneous computing devices, architectures for programmable, reconfigurable logic (FPGA), and HW-SW co-design of embedded systems. In these areas, he has published more than 30 papers in international peer-reviewed conferences and journals, with more than 1000 citations and an h-index of 15.

Alessandro Biasci is a team leader at Evidence SRL. He previously graduated in Computer Engineering at the University of Pisa. His research interest is mainly on the design of resource-scheduling techniques for embedded and real-time systems, with specific focus on memory subsystem. He was involved in national and european research projects.

Paolo Valente is an assistant professor at the Department of Computer Science of the University of Modena and Reggio Emilia, Italy. His research activity focuses mainly on designing and analyzing resource-scheduling algorithms. Several of his algorithms have in been included in mainstream operating systems (Linux, FreeBSD, OS X). He was and is involved in national and european research projects.

Andrea Marongiu received the Ph.D. degree in Computer and Electronic Engineering from the University of Bologna, Italy, in 2010. He has been a postdoctoral research fellow at ETH Zurich, Switzerland. He currently is an associate professor at the University of Modena and Reggio Emilia. His research interests focus on programming models and architectures in the domain of heterogeneous multi- and many-core systems-onchip. In this field, he has published more than 120 papers in international peer-reviewed conferences and journals.