UNIMORE

UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dottorato di ricerca in

COMPUTER AND DATA SCIENCE FOR
TECHNOLOGICAL AND SOCIAL INNOVATION

Ciclo XXXVIII

Taming unpredictable timing in heterogeneous SoCs via whole-system memory bandwidth regulation

Candidato	Dott. Lorenzo Carletti
Relatore (Tutor):	Prof. Andrea Marongiu
Correlatore (Co-Tutor):	Prof. Paolo Valente
Coordinatore del Corso di Dottorato:	Prof. Andrea Marongiu

Abstract

Heterogeneous Systems on Chip (HeSoCs) combine the benefits of multicore CPU performance with efficient high-throughput accelerators. Commercial Off the Shelf (COTS) HeSoCs are readily available IPs that system integrators can use to deploy their products at low cost. In most COTS HeSoCs, different compute units (CUs) – like the main CPU cores and one or more accelerators – share parts of the memory hierarchy, typically the last-level cache (LLC) and the main DRAM memory. In such systems the memory hierarchy is normally designed for best-effort performance, which might introduce unpredictable timing behavior for memory-intensive workloads (scenarios where the CUs cumulatively request up to 3 times the available bandwidth are not uncommon in modern HeSoCs). System bandwidth saturation ultimately translates in unpredictable slowdown of the tasks running. While this might be tolerable for general-purpose compute systems, this unpredictability makes the adoption of COTS HeSoC difficult in time-critical domains.

Various memory interference mitigation techniques have been explored to tackle the problem, but they mostly focus on controlling individual components. We advocate that memory interference mitigation must account for the simultaneous operation of different types of CUs to properly reduce task slowdowns.

First, in this thesis we present an exploration of how memory interference is generated in the memory hierarchy of modern HeSoCs. In particular, we differentiate between the various levels of the memory hierarchy at which interference can happen. CPU cores can access shared caches and the main memory, so they can interfere each other at multiple levels. We highlight three main causes for interference: i) true main memory sharing (interference); ii) mutual cache conflict contention (cache evictions due to conflict misses in shared cache levels); iii) contention for hardware cache resource sharing in absence of evictions (cache congestion interference). Accelerators

typically only interfere at the LLC and main memory level, but they are capable of generating much higher memory bandwidth requests compared to CPU cores, so they end up causing disproportionate slowdowns to CPU cores.

Then, we introduce a control scheme capable of reducing the slowdown experienced by tasks running on COTS HeSoCs via memory bandwidth throttling. The first version of the control scheme combines an offline bandwidth characterization with online bandwidth probing to estimate how throttling the accelerators affects the slowdown that CPU tasks are subject to. It then throttles the accelerators to match user requested maximum slowdown. With this setup, we observe that we are able to guarantee maximum slowdown requests with near-zero error (2%), while at the same time throttling the accelerators by the minimum amount that provides this guarantee, thus maximizing system bandwidth utilization.

Finally, we present an improved version of the control scheme capable of also throttling the CPU cores. Thanks to this, and the previous memory interference exploration, we are able to implement a control scheme based on a gradient descent technique. This improved control scheme is able to reduce the slowdown that a CPU critical task is subject to on HeSoC by throttling both the accelerators and the CPU cores.

Acknowledgments

In this section, I would like to address all the people which were an important part of these three years as a PhD student.

First and foremost, I would like to thank my supervisors: Andrea Marongiu and Paolo Valente. They introduced me to the topic that became my PhD project, and they guided me along the way. Their knowledge, their feedback, and the back-and-forth of ideas we had on a weekly basis shaped this journey.

Then, I would like to thank Alessandro Capotondi and Gianluca Brilli for both their technical support and our repeated collaborations throughout the years, as well as some laughs.

I would also like to thank Björn Forsberg and RISE for hosting me for various months. In a small yet productive span of time we were able to really dig deep into the topic we focused on, with significant results.

After that I would like to thank my PhD peers: Andrea Serafini, Federico Motta and Andrea Artioli. While we did collaborate at a professional level, we also had copious amounts of laughs, which made the three years significantly better.

I also wish to thank my friends, for the good times we had during these years. Like when going out to eat dinner, or when discussing the weirdest things possible during the evening.

Lastly, and most importantly, I wish to thank my family for their full and unconditional support through these years. I literally could not have done it without them.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Contributions . . . 4

1.2.1 Memory interference exploration . . . 5

1.2.2 Characterization-based Bandwidth Manager - Accelerators . . . 6

1.2.3 Characterization-based Bandwidth Manager - Accelerators and CPUs . . . 6

1.3 Structure of the thesis . . . 7

2 Background **8**

2.1 The rise of heterogeneous systems . . . 8

2.2 HeSoC - Architectural template . . . 9

2.2.1 CPU complex . . . 9

2.2.2 Accelerator complex . . . 10

2.2.3 Memory subsystem . . . 10

2.2.3.1 Caches . . . 11

2.2.3.2 Main memory . . . 11

2.2.3.3 System interconnect . . . 12

2.3 Memory Interference . . . 12

2.3.1 Cache partitioning . . . 14

2.4 Relevant platforms . . . 15

2.4.1 *Xilinx ZU9EG* . . . 15

2.4.2 *Nvidia TX2* . . . 16

2.4.3 *Nvidia AGX Orin* . . . 17
2.4.4 *Raspberry Pi 4* . . . 18
2.4.5 *Raspberry Pi 5* . . . 19
2.5 Workloads . . . 19
2.5.1 SynthMemBench synthetic benchmarks . . . 20
2.5.2 *PartitionedChaser* synthetic benchmark . . . 24
2.5.3 The Polybench-ACC benchmark suite . . . 28

3 Related work **30**
3.1 Memory interference exploration . . . 30
3.1.1 Platform-specific memory characterization . . . 30
3.1.2 DRAM maximum interference estimate . . . 31
3.1.3 Cache interference characterization . . . 31
3.2 Memory interference mitigation . . . 32

4 Memory interference exploration **35**
4.1 Main memory interference . . . 35
4.1.1 Analysis Summary . . . 39
4.1.2 Comparison with the literature . . . 40
4.2 Cache conflict contention . . . 41
4.2.1 *CpuR* interfered by *CpuR* - Refills . . . 42
4.2.2 *CpuR* interfered by *CpuW Miss* - Refills and Writebacks 45
4.2.3 *CpuR* interfered by *CpuW* - Filling the write buffer . . 47
4.2.4 Analysis Summary . . . 49
4.3 Cache congestion . . . 50
4.3.1 Polybench Cache Congestion Slowdown . . . 54
4.3.2 Comparison with the literature . . . 56

5 Characterization-based Bandwidth Manager **58**
5.1 Target Platform and System Model . . . 61
5.1.1 Hardware platform . . . 61
5.1.2 The throttling factor . . . 62
5.1.3 System model . . . 63
5.2 The Bandwidth Manager (*CBM*) . . . 65

5.2.1 Offline phase 67
5.2.1.1 **Traffic generators** 68
5.2.1.2 **Data structure and characterization algorithm** 69
5.2.1.3 **Shape and properties of CPU and accelerator bandwidth curves** 69
5.2.2 Online Phase 74
5.2.2.1 **Determining the optimal *Throttling Factor*** 76
5.2.2.2 **Correcting the target bandwidth** 82
5.3 Experimental Evaluation 84
5.3.1 *CBM* implementation, overhead and time granularity . 86
5.3.2 MTS guarantee - Polybench 87
5.3.3 MTS guarantee - Speed 90

6 HeSoC-wide gradient-based Bandwidth Manager **93**
6.1 Memory interference in HeSoCs - Accelerators and CPU cores 94
6.2 Expanding the bandwidth manager 97
6.2.1 Bandwidth throttling the CPU cores 97
6.2.2 A gradient-based bandwidth manager 98
6.3 Experimental evaluation 101

7 Conclusions and Future work **104**
7.1 Contribution Summary 104
7.2 Limitations and Future Work 105
7.3 Practical implications 106
7.4 Concluding remarks 106

Chapter 1

Introduction

1.1 Motivation

Nowadays, Systems on Chip (SoC) are widely used in everyday scenarios for their embedded computing capabilities. In particular, Heterogeneous Systems on chip (HeSoC) have been getting increasingly adopted for various workloads. HeSoCs combine multi-core CPUs, useful for their computing flexibility, with accelerators, specialized hardware that can increase the throughput of specific tasks while keeping low power consumption. This combination of compute units (CUs) allows to execute both generic tasks and High Performance Computing (HPC) on small embedded devices, making HeSoCs attractive for system integrators targeting applications that have strict size, weight and power (SWaP) requirements (e.g. autonomous driving, autonomous drone piloting, wearable devices...). Some manufacturers sell Commercial-Off-the-Shelf (COTS) HeSoCs, which are HeSoCs containing a predetermined set of hardware. While they are not tailor-made for any specific workload, they usually target an average use-case chosen by the vendor. Thanks to this, they are generally a much cheaper alternative to designing and manufacturing an HeSoC from the ground up.

Figure 1.1 shows the template of a modern COTS HeSoC. To keep both costs and complexity low, manufacturers typically make various levels of the memory hierarchy shared (as can be seen in Figure 1.1, with the cluster cache,

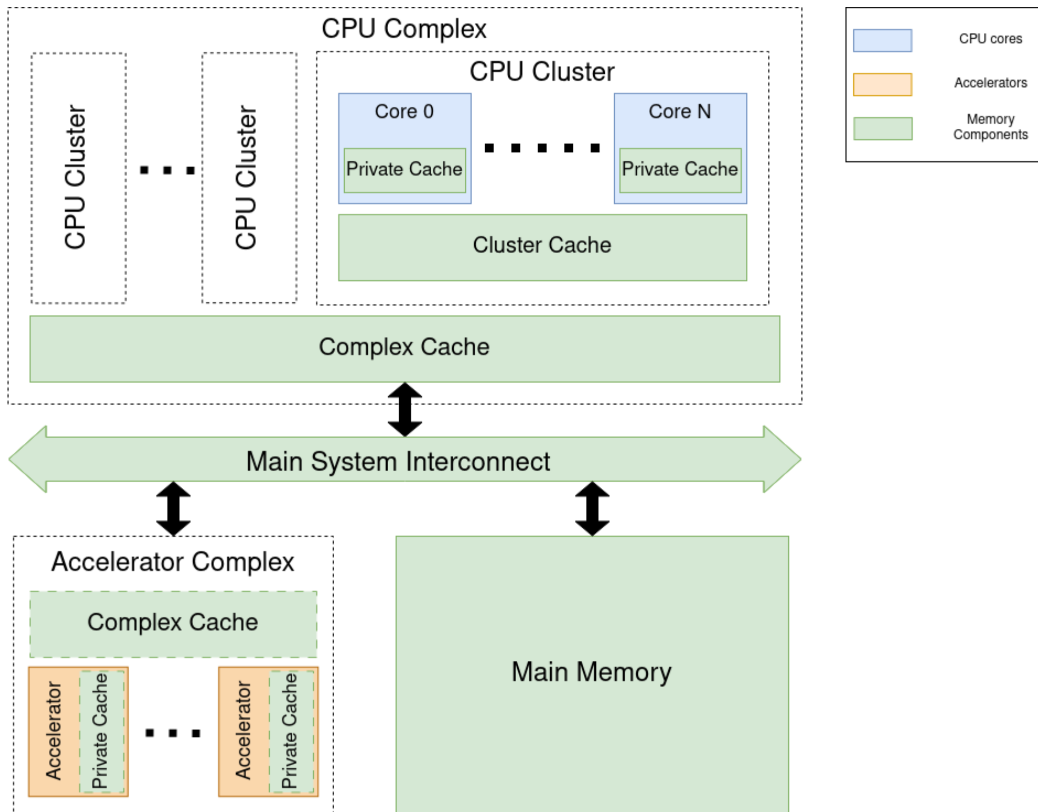


Figure 1.1: Modern COTS HeSoC template. The accelerator complex may or may not have caches.

the complex cache, the main system interconnect and the main memory).
However, as the number of CUs in the systems grows, the shared memory components are subject to an increased amount of contention under demanding scenarios. Because of this contention (also known as *memory interference*), tasks may experience an increase in the latency of their memory operations (i.e. a decreased bandwidth towards the shared memory resources) and consequently an increase in execution time (a *slowdown*) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. As an example, on two of our reference HeSoCs, the Xilinx ZU9EG and the NVIDIA Orin AGX, when all CUs try to access the main memory at the same time, we see slowdowns for a CPU task of up to 7× and 4.5× respectively.
Best-effort applications may tolerate occasional, unbounded slowdowns. In

contrast, a *time critical* task usually has Quality-of-Service (QoS) requirements, and fails to meet these requirements if it suffers from more than some maximum tolerated slowdown. This seriously limits the adoption of COTS HeSoCs in certain environments (e.g. real-time). A solution to this problem could be to use a more powerful COTS HeSoC to have the task still meet the QoS requirements thanks to performance improvements. However, this solution may not be viable because either: i) the SWaP requirements of the application would be exceeded; or ii) the *time critical* task may still not meet the QoS requirements, due to resource sharing causing high memory interference even on newer platforms. An alternative solution could be to redesign the HeSoC, to reduce the amount of memory hierarchy sharing, but that has the downsides already explained before. An actually viable solution to have *time critical* tasks meet QoS requirements on COTS HeSoCs is memory interference mitigation.

Memory interference mitigation has been extensively studied before. Both on COTS HeSoCs [17, 18, 19, 20, 21, 22, 10, 23] and on other kinds of platforms [24, 25, 26, 27, 28].
One possibility is removing the memory interference entirely. This can be achieved by having only one CU at a time may access the contended memory component [10, 27, 23]. The problem with such an approach is that it underutilizes the memory bandwidth [22]. While this is guaranteed to work, and provide a 1.0× slowdown, underutilizing the COTS HeSoC can lead to requiring more powerful hardware, as well as decreased efficiency.
Another widely adopted solution is bandwidth regulation [24, 29, 25, 18, 19, 20, 21, 26]. In particular, bandwidth regulation reduces the amount of memory interference that a CU causes by lowering the amount of memory bandwidth that it can demand, through throttling of memory accesses. This, in turn, can be used to lower the amount of slowdown that *time critical* tasks are subject to, and guarantee QoS requests.
Most bandwidth regulation approaches are only designed to work on either the CPUs [24, 29, 25] or the accelerators [18, 19, 20, 21, 26]. However, for bandwidth regulation to be effective on COTS HeSoCs, the bandwidth of both the CPUs and the accelerators must be regulated [17].

These solutions already present in the literature can be combined in COTS HeSoC to handle QoS requirements for pre-known workloads (in a setting like avionics [17], as an example). However, ensuring proper bandwidth regulation (without hardware underutilization) for generic (not pre-known) workloads for a given QoS is unexplored when it comes to COTS HeSoCs. This is because understanding how much to throttle each CU for a QoS is not easy, as how much memory interference a CU is causing heavily depends on the task being executed. Not only that, but tasks of generic workloads may change their bandwidth usage (meaning both how much interference they cause and how much they are slowed down by other tasks) at any time during execution, making it even more difficult to find how much to throttle each CU.
This is the gap that this thesis aims to fill: a way to guarantee the QoS for a *time critical* task of a generic workload that can change bandwidth usage at runtime via bandwidth regulation of both the CPU and the accelerators on COTS HeSoCs.

1.2 Contributions

In this chapter, the contributions of this thesis are introduced. To guarantee the QoS for a *time critical* task via bandwidth regulation on COTS HeSoCs, first we start with a *memory interference exploration*, to understand how interference is generated in the memory hierarchy of modern HeSoCs. This is further explained in Section 1.2.1. Then, in Section 1.2.2 we present *Characterization-based Bandwidth Manager* (*CBM*), a bandwidth manager capable of guaranteeing the QoS for a *time critical* task via bandwidth regulation of accelerators on COTS HeSoCs. Finally, in Section 1.2.3 we introduce an upgraded version of *CBM*, capable of also throttling CPU cores in COTS HeSoCs, so that regardless of the workload under execution, the QoS of the *time critical* task is guaranteed.

1.2.1 Memory interference exploration

In this contribution, we study how memory interference is generated in modern HeSoCs: at what level of the memory hierarchy it can happen, which traffic type is the most sensitive to interference, and which traffic type is the one that causes the highest amount of interference. As shown in Figure 1.1, there are multiple shared memory components inside of a modern HeSoC, and interference affects them differently. To have comparable results between the different types of memory interference, we do our exploration only using CPU cores, as they have access to all the shared memory components. In general, what we observe is that the main memory and the main system interconnect can be subject to temporal interference due to all the CUs of the system accessing them. We measure the amount of memory interference by having all CPU cores running workloads that target the main memory.
After that, we study interference on the shared caches. In general, we observe that shared caches can be subject to both spatial interference (mutual cache conflict contention) and temporal interference (cache congestion interference). Mutual cache conflict contention happens when the data of a victim task is evicted from the shared caches due to accesses of other CPU cores. When this happens, the victim task will need to do extra cache refills (e.g. from the main memory) to get the data back to the shared cache level, leading to an increase in execution time. Cache congestion interference happens when multiple cores use the cache at the same time, without any operation targeting lower memory hierarchy levels (like the main memory). To measure this kind of memory interference, we introduce a synthetic benchmark with which we differentiate between spatial interference and temporal interference in caches, and we execute it on all CPUs sharing the target cache.
The results of this exploration show that: i) read intensive CPU tasks targeting the main memory are not the ones subject to the highest slowdowns possible or causing the highest levels of interference, disproving common misconceptions on the matter [30, 14, 22, 16, 15, 31] ; ii) mutual cache conflict contention can cause even higher slowdowns than main memory interference, with possible causes being data evictions, cache maintenance opera-

tions caused by other cores, or hardware limitations (like the write buffer of the cache being filled by interfering cores) ; iii) cache congestion can also cause significant slowdowns, despite not being a focus in the literature [4].

1.2.2 Characterization-based Bandwidth Manager - Accelerators

In this contribution, we present *Characterization-based Bandwidth Manager* (*CBM*), a bandwidth manager that regulates the bandwidth of main-memory accesses of accelerators, for generic workloads that may change their bandwidth usage at any time during execution, without knowing either the total execution times of tasks, or the fractions of execution time that tasks devote to memory accesses. *CBM* regulates the bandwidth of only the accelerators, by computing and enforcing a global *throttling factor*. First, during an offline phase, *CBM* performs a characterization of how a generic CPU task may be slowed down on the platform at hand, as a function of the memory traffic generated by the task itself and by the accelerators (inspired by what is done during the *memory interference exploration*). Then, at runtime, *CBM* uses this characterization to guess a candidate throttling factor, in one step, based on the bandwidth measured for the *time critical* task and the accelerators. Finally, *CBM* employs a runtime periodic check that handles small bandwidth variations by slightly changing the throttling factor, and big bandwidth (or traffic type) variations by recalculating the candidate throttling factor.

We observe that even under the most demanding conditions in our experiments, *CBM* is able to guarantee the QoS requirements of *time critical* tasks accurately, with at worst a 2% error.

1.2.3 Characterization-based Bandwidth Manager - Accelerators and CPUs

For the final contribution, we added to *CBM* the ability to throttle CPU tasks as well. In general, this grants *CBM* the ability to throttle all CUs in

an HeSoC, achieving the goal of the thesis.

In particular, thanks to the *memory interference exploration* contribution, we know how the different types of memory interference can combine. We use this knowledge to create an initial bandwidth manager, based on a gradient descent technique, that is capable of deciding how to throttle both the CPU cores and the accelerator.

1.3 Structure of the thesis

The structure of the thesis is as follows: Chapter 2 presents the background information necessary to understand the thesis, the hardware used for experiments, and concepts that are shared across contributions; Chapter 3 collocates the thesis with respect to the literature, highlighting how the work we discuss is different from previous works; Chapter 4 exposes the problem of memory interference on HeSoCs and catalogs the various types of memory interference that are possible, highlighting both a underestimated type of cache interference as well as common pitfalls present in the literature related to memory interference in general; Chapter 5 presents *CBM* and evaluates its performance at guaranteeing execution times for synthetic and real-world tasks; Chapter 6 proposes the improved version of *CBM*, capable of throttling both CPU cores and accelerators, and measures its capabilities; Chapter 7 presents the final remarks as well as the direction of future work.

Chapter 2

Background

In this Chapter, we describe the background needed to understand this thesis. First, modern HeSoCs are described, as well as their architectural template, with a focus on the memory subsystem. After that, we describe in detail the platforms used in the thesis, along with their features and peculiarities. Then, we explain the phenomenon of memory interference, along with common causes. Finally, we present the synthetic benchmarks we use in all subsequent chapters, with algorithms that explain how they work.

2.1 The rise of heterogeneous systems

During the early years of computing, performance improvements came from increases to the clock frequency of single core CPUs. However, by the early 2000s increasing the clock frequency further became impossible, due to power consumption and heat generation of these cores not scaling linearly with clock frequencies [32]. Multicore CPUs were chosen as the alternative, thanks to the ability to grant comparable performance levels at lower clock rates. However, while generic multi-core CPUs are able to execute many tasks via software, there are some workloads that can be optimized by using specialized hardware accelerators instead (e.g. image manipulation, or inference). In particular, accelerators are used to improve the speed with which certain tasks are completed, while keeping the total power consumption (and heat

generation) of the system down. Nowadays, heterogeneous embedded computing, i.e. combining multicore CPUs with hardware accelerators, is the norm in the industry, thanks to all the applications it enables.

2.2 HeSoC - Architectural template

Figure 1.1 shows a generic block diagram of a modern COTS HeSoC, highlighting its three main architectural blocks: *CPU complex*, *Accelerator complex*, and the *Memory subsystem* (which includes both the main memory as well as the caches present in the complexes).

2.2.1 CPU complex

The CPU complex (or host complex) is typically composed of one or more *Central Processing Unit* (CPU) cores, which can be organized into homogeneous clusters. CPU cores are general-purpose, and they can execute many different operations (as defined by their *Instruction Set Architecture*, or ISA). Because of this, the CPU complex is used in HeSoCs to host the main application logic, to handle system resource management (e.g. running an operating system in the background), and to coordinate accelerator tasks. Modern CPU cores have one or more levels of private caches. These caches store frequently used data and instructions, such that future accesses have a lower latency (when compared to reading the data from the main memory). CPU cores in a cluster can share a unified cluster cache. Moreover, if there are multiple CPU clusters, all CPU cores in the CPU complex can share a complex cache. In general, the depth and complexity of the cache hierarchy varies with the number of cores and their clustering. These typically bigger shared cache levels help with both reducing access latency and efficient data sharing amount cores. 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.

2.2.2 Accelerator complex

The accelerator complex of an HeSoC is composed of one or more specialized hardware units (accelerators) designed to only execute specific workloads. In general, accelerators trade the ability to execute many operations (typical of general-purpose CPU cores) for domain-specific architectural optimizations useful to perform certain tasks. These optimizations are what grant accelerators higher throughput and efficiency than CPU cores for specific workloads. Common accelerator types found in HeSoCs include General-Purpose Graphics Processing Units (GPGPUs) for heavily data-parallel workloads, Digital Signal Processors (DSPs) for signal processing (e.g. audio or image processing), and Field Programmable Gate Arrays (FPGAs) for fast development of new hardware architectures tightly coupled with custom software on the host side. Modern accelerators are typically paired with *Direct Memory Access* (DMA) engines. DMA engines are hardware components specialized in moving data to and from the main memory. DMA engines are also able to concurrently provide data to accelerators while these are executing workloads offloaded from the host side, hiding the latency caused by data movements. Modern accelerators may also include caches and scratchpad memories to reduce the time needed to access specific data, further improving their performance.

2.2.3 Memory subsystem

The memory subsystem of HeSoCs stores data required by the compute units, and allows them to efficiently communicate with one another. It is typically organized in a hierarchy, with the upper levels (i.e. the ones closer to the compute units) being faster to access, and the lower levels having a significantly higher capacity, at the cost of an increased access latency. The main components of the memory subsystems are the caches, the main memory, and the system interconnect, which routes the data across all other components.

2.2.3.1 Caches

Caches are high-speed memory components designed to temporarily store data recently accessed by compute units for rapid retrieval. They operate on the assumption that compute units are likely to access: i) the same memory locations multiple times (temporal data access locality, i.e. accessing the same variable multiple times); ii) memory locations close to one another (spatial data access locality, i.e. accessing consecutive values in an array). To reduce latency, they are also typically positioned close to the compute units, further reducing the physical time needed to access the data. Thanks to all these optimizations, caches reduce the access latency of most memory operations, significantly improving overall system performance in modern systems (as the main memory access latency spans hundreds of clock cycles).

The caches present in modern HeSoCs are typically *n-way set associative*. A *n-way set associative* cache is internally organized as a matrix of cache lines. Cache lines are the basic unit of transfer between a cache and the other memory hierarchy levels. In *n-way set associative* caches, the matrix of cache lines is organized in sets (horizontal rows) and ways (vertical columns). Cache lines internally also have some metadata. For the purposes of this thesis, the most important metadata are the *tag*, used to identify the address in the main memory that a cache line corresponds to, and the *dirty bit*, used to track whether the data has been altered by one of the compute units using it.

2.2.3.2 Main memory

The main memory is the primary volatile storage for data used by the compute units of an HeSoC. The main memory is usually implemented using the *Dynamic Random Access Memory* (DRAM) technology. Internally, DRAM is structured in independent data *banks*, the memory modules that store the data. Different data banks can be accessed separately, allowing *Bank-Level Parallelism* and improving performance.

The memory controller manages accesses to the main memory. This component of the main memory, connected to the system interconnect, satisfies

the memory requests coming from the various actors of the systems (e.g. CPUs, GPGPUs, DMA engines...). The memory controller can combine and schedule requests to optimally use the banking structure of the DRAM, reducing the latency experienced by the actors of the system.

2.2.3.3 System interconnect

The main system interconnect is the communication channel linking compute units and peripheral controllers to the main memory. The main system interconnect is able to sort and forward memory requests to the main memory. In particular, the main memory serves the incoming requests (acting as a *slave*), while the other actors of the systems generate them (acting as *masters*). While older computing systems used shared buses or point-to-point connections to communicate with the main memory, these topologies hit severe performance and scalability bottlenecks as the number of compute units in systems increased [33]. For this reason, modern HeSoC typically use *crossbar-switch* architectures or *Network-on-Chip* (NoCs) [34]. Crossbar architectures use a matrix-based topology to connect directly any possible master to any possible slave, with a switch that multiplexes requests. In NoCs, instead, masters and slaves send and receive network packets by using standard communication protocols (like AMBA AXI or CHI [35]). A fabric of routers and links then allows the packets to reach their destination. Both architectures enable concurrent data transfers from multiple actors, satisfying the performance and scalability requirements of modern HeSoCs.

2.3 Memory Interference

The main interconnect and main memory controller of HeSoCs 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. 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 a key 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. While private caches are immune to interference, as they are not shared among cores, cluster caches or lower-level caches shared by more than one core may also be subject to interference.

In particular, for shared caches there are multiple possible causes of memory interference. The most common type is *line conflict* [1, 2]. As multiple cores use the cache, they can evict the data of each other. When this happens, victim cores will need to look for the data in the lower memory hierarchy levels and place it back in the cache (line *refill*), incurring extra time penalty.

Write operation can also create interference. If *writeback* is the policy implemented for stores, the cache line is flagged as *dirty*, but its content is not immediately propagated to the lower memory hierarchy levels. Only when a *dirty* line is evicted the write operation to the lower levels is executed, and only at that moment the core causing the eviction incurs the time penalty. If one core commits a store but doesn't evict the corresponding cache line, a victim core could end up doing the writeback operation for them, causing interference.

While modern *n-way set associative* caches reduce the frequency of evictions for the common case (by ensuring one address can map to n different cache lines), the amount of space in a cache is still limited, so memory intensive workloads can still evict each other.

Finally, cache interference can be caused by too many concurrent requests. This can happen because of high pressure on certain key registers in operations like refills or writebacks (e.g. MSHR [4] and WriteBack Buffer [12]), or because the cache itself is unable to satisfy all the requests

coming from the higher memory hierarchy levels (cache *congestion*).

All these different types of interferences can combine with one another, creating the potential for exceptional slowdowns [12].

We explore this further with platform characterizations in Section 4.

2.3.1 Cache partitioning

Cache partitioning is a common technique to reduce memory (cache) interference. We use cache partitioning in one of the synthetic benchmarks we created for this thesis (introduced in Section 2.5.2). Cache partitioning organizes the cache in subsets [36, 6] that can be individually assigned to compute units to isolate task execution, preventing line conflicts. While it limits how much of the cache a task is able to use, reducing this type of conflicts can lead to lower and more consistent execution times, even as other compute units also make use of the cache.

Some modern caches have builtin partitioning hardware [37, 38], which does not require to modify the software executing on the CPUs. Software-based cache partitioning is an alternative which requires special memory allocation to enforce the partitioning. To account for the different usage of the memory, software-based cache partitioning requires using hypervisors, modifying the OS, or recompiling the software [6]. Cache coloring, an eviction mitigation technique which ensures that buffers allocated by tasks make use of contiguous sets in the cache, also falls into the category of software-based cache partitioning [39, 40, 41, 6].

Cache partitioning can be done with different *modes* [6]. Way-based *modes* assign cache *ways* to cores. This can easily be implemented in hardware, especially with modern shared caches which tend to have high associativity. Set-based *modes* instead assign entire cache *sets* to compute units. While very granular, it combines especially well with software based cache partitioning, as it is not possible to target in which way the data is placed without hardware support. Block-based *modes* assigns specific *blocks* to cores. It is the most granular level of cache partitioning and requires the highest level of hardware and software support.

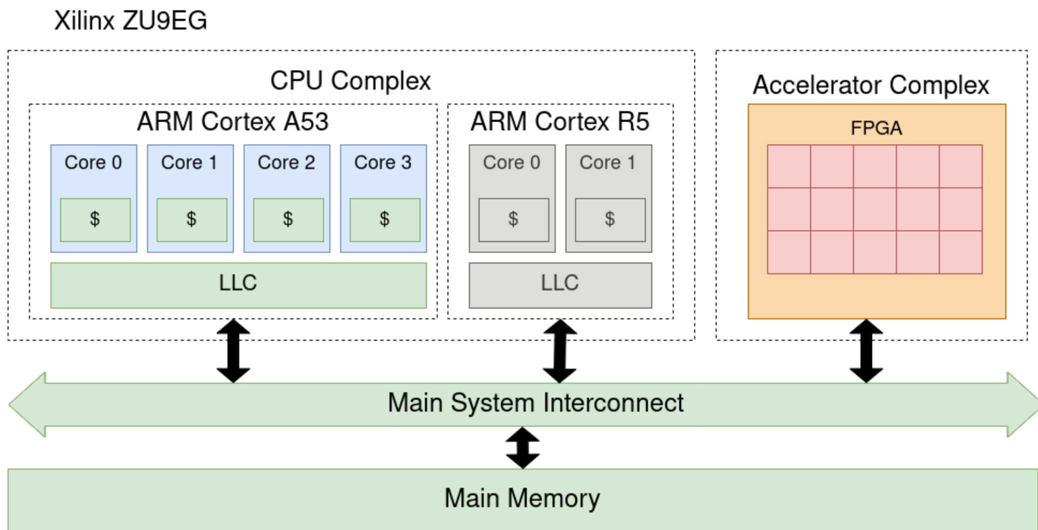


Figure 2.1: Architecture of the *Xilinx ZU9EG*.

2.4 Relevant platforms

Most modern COTS HeSoC adhere to the architectural template described in Section 2.2. In this section, we present the platforms used in this thesis in detail. In particular, we discuss the following HeSoCs: the *Xilinx ZU9EG*, the *Nvidia TX2* and the *Nvidia AGX Orin*. We also present two MSoCs that we use as data points for our memory interference exploration in Section 4.3: the *Raspberry Pi 4* and the *Raspberry Pi 5*. We run the Linux kernel as the background operating system on all these platforms.

2.4.1 *Xilinx ZU9EG*

The *Xilinx ZU9EG* is part of the UltraScale+ family of heterogeneous SoCs produced by AMD [42].

Figure 2.1 shows the architecture of the *Xilinx ZU9EG*. The *Xilinx ZU9EG* features two CPUs: a quad-core ARM Cortex A53 general-purpose CPU, and a dual-core ARM Cortex R5 real-time processing unit. For the purposes of this thesis, we mainly focus on the main host CPU, the ARM Cortex A53 CPU. Each core of the ARM Cortex A53 CPU features a 32 KB private 2 way set associative L1 instruction cache (L1 ICache) and a 32 KB private 4

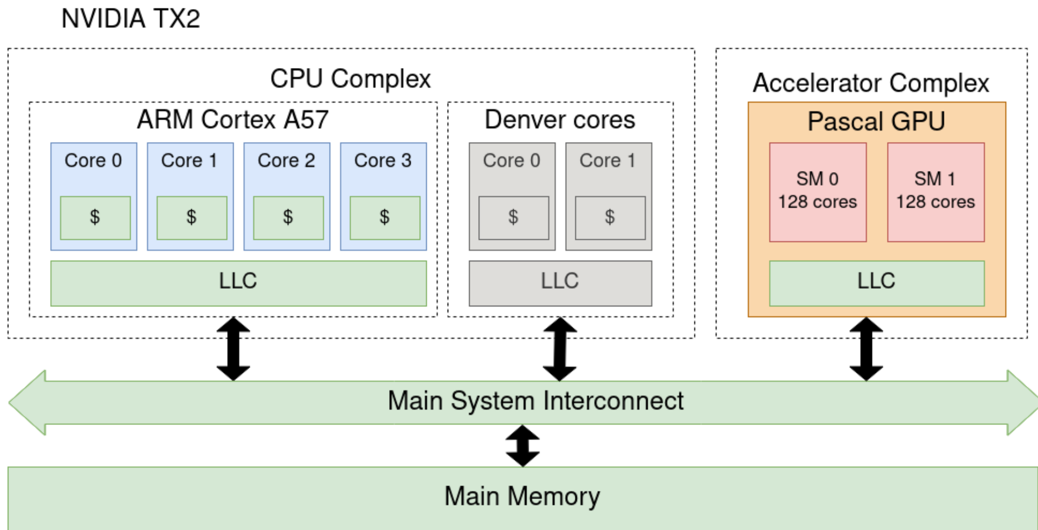


Figure 2.2: Architecture of the *Nvidia TX2*.

way set associative L1 data cache (L1 DCache). The four ARM Cortex A53 CPU cores share a 1 MB 16 way set associative L2 cache. All caches use 64 bytes cache lines.

The accelerator complex of the *Xilinx ZU9EG* hosts a FPGA with 274 thousand LUTs, twice the amount of Flip-Flops and 4 MBs of block RAM. The main memory of the *Xilinx ZU9EG* is a 4 GB LPDDR4 memory with a 128 bit bus width, operating at 1.3 GHz.

Section 5.1.1 presents further details on the *Xilinx ZU9EG*, its FPGA and the way the actors of the platform are connected to the Main Memory. Those hardware details are presented separately, as needed. This is because, alongside the hardware, Section 5.1.1 also presents details on how the FPGA is specifically instantiated for Section 5. Similar discussions are not present for the other platforms.

2.4.2 *Nvidia TX2*

The *Nvidia TX2* is a NVIDIA HeSoC released in 2017 [43].

Figure 2.2 shows the architecture of the *Nvidia TX2*. The *Nvidia TX2* features two CPU clusters: a cluster composed of four ARM Cortex A57 cores, and a more powerful cluster composed of two *Denver* cores (ARM-

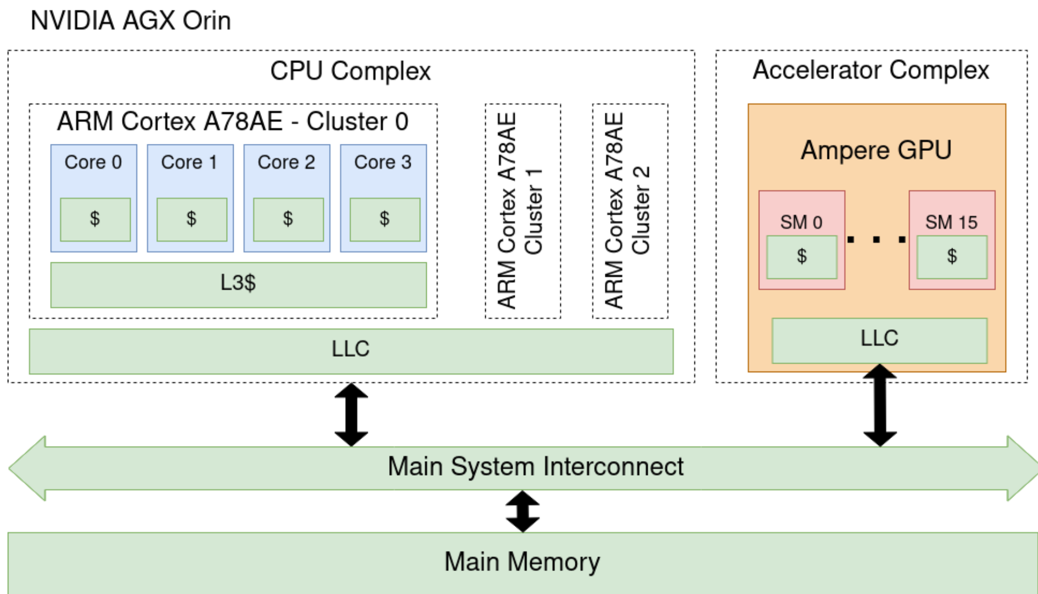


Figure 2.3: Architecture of the *Nvidia AGX Orin*.

compliant cores designed by NVIDIA). For the purposes of this thesis, we mainly focus on the ARM Cortex A57 cluster. Each core of the ARM Cortex A57 cluster features a 48 KB private 3 way set associative L1 ICache and a 32 KB private 2 way set associative L1 DCache. The ARM Cortex A57 cluster shares a 2 MB 16 way set associative L2 cache. All caches use 64 bytes cache lines.

The accelerator complex of the *Nvidia TX2* hosts a GPU based on the *Pascal* architecture, featuring 2 SMs with 128 cores each. The main memory of the *Nvidia TX2* is an 8 GB memory with a 128 bit bus width, operating at 1.866 GHz.

2.4.3 *Nvidia AGX Orin*

The *Nvidia AGX Orin* is a NVIDIA HeSoC released in 2022 [44].

Figure 2.3 shows the architecture of the *Nvidia AGX Orin*. The *Nvidia AGX Orin* features three homogeneous CPU clusters, each formed by four ARM Cortex A78AE cores, for a total of 12 CPU cores. Each core features a 64 KB private L1 ICache and a 64 KB private L1 DCache, as well as a 256

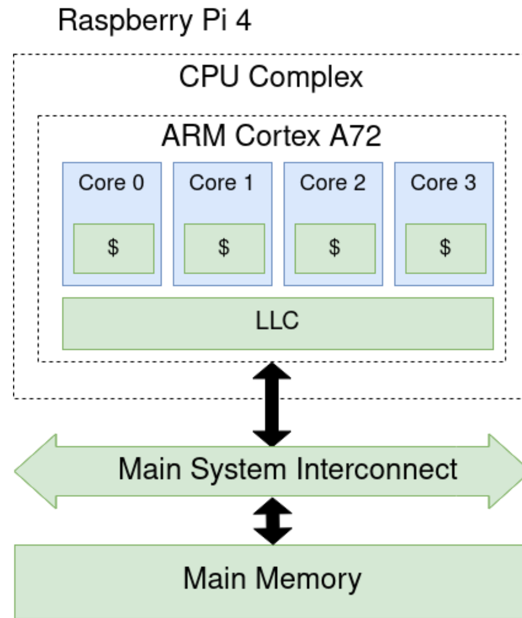


Figure 2.4: Architecture of the *Raspberry Pi 4*.

KB private 8 way set associative L2 cache. Each core in a cluster shares a 2 MB 16 way set associative L3 cache, with a global 4 MB complex cache. All caches use 64 bytes cache lines.

The accelerator complex of the *Nvidia AGX Orin* hosts a GPU based on the *Ampere* architecture, featuring 16 SMs with 128 cores each. Each SM has 192 KB of L1 cache, with a shared 2 MB cache among all SMs. The main memory of the *Nvidia AGX Orin* is a 64 GB LPDDR5 memory with a 256 bit bus width, operating at 3.2 GHz.

2.4.4 *Raspberry Pi 4*

The *Raspberry Pi 4* is an MSoC released in 2019 [45].

Figure 2.4 shows the architecture of the *Raspberry Pi 4*. The *Raspberry Pi 4* features one CPU cluster, formed by four ARM Cortex A72 cores. Each core features a 48 KB private 3 way set associative L1 ICache and a 32 KB private 2 way set associative L1 DCache. Each core shares a 16 way set associative 1 MB L2 cache. All caches use 64 bytes cache lines.

The main memory of the *Raspberry Pi 4* is a 8 GB LPDDR4 memory,

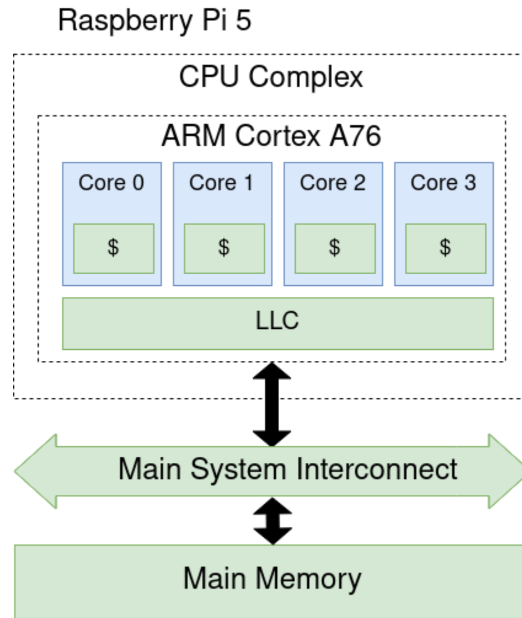


Figure 2.5: Architecture of the *Raspberry Pi 5*.

operating at 2.4 GHz.

2.4.5 *Raspberry Pi 5*

The *Raspberry Pi 5* is an MSoC released in 2023 [46].

Figure 2.5 shows the architecture of the *Raspberry Pi 5*. The *Raspberry Pi 5* features one CPU cluster, formed by four ARM Cortex A76 cores. Each core features a 64 KB private L1 ICache and a 64 KB private L1 DCache, as well as a 512 KB private 8 way set associative L2 cache. Each core shares a 2 MB 16 way set associative L3 cache. All caches use 64 bytes cache lines.

The main memory of the *Raspberry Pi 5* is a 16 GB LPDDR4X memory, providing up to 17 GB/s of memory bandwidth.

2.5 Workloads

In this section, the workloads used for the experiments presented in this thesis are described.

2.5.1 SynthMemBench synthetic benchmarks

Synthetic memory-access generators are commonly used to produce controlled interference and measure resulting slowdowns [30, 14, 22, 16, 15, 31]. They allow a wide spectrum of access patterns to be easily generated. In general, a synthetic benchmark [47, 48] 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 *CpuW* 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 *CpuW Miss* type, represented by a single store) due to the significantly different behaviour 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. In particular, we define the memory intensity $I \in [0, 1]$ of a task as the fraction of the total execution time, in isolation (i.e. without any other task running on the other CPU cores of the platform), that the task spends executing memory accesses. A compute-bound or memory-bound task has then a memory intensity equal to 0 or 1, respectively. It should be noted that *cops* is a platform-dependent number, which depends on factors like memory access speeds and CPU clock speeds. In general, the number should be tuned according to the target platform to emulate specific memory-to-compute ratios (e.g. 100-0, 50-50, or 25-75).

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* [13, 30], 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 [13, 30];

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 one set of synthetic benchmarks that we designed for this thesis [49]. 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 *CpuR*, *CpuRW*, *CpuW Miss* and *CpuW*. The four 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 four (*CpuR* has $t = r$, *CpuRW* has $t = rw$, *CpuW Miss* has $t = ws$, and *CpuW* has $t = w$).

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

```
1
2  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
3      i ← 0;
4      offset ← 0;
5      while i < mops do
6          if t = r then
7              load(addr + offset)
8          end
9          if t = w then
10             for i ← 0 to LINE_SIZE by 4 do
11                 store(addr + offset + i)
12             end
13         end
14         if t = rw then
15             for i ← 0 to LINE_SIZE by 4 do
16                 load(addr + offset + i)
17                 store(addr + offset + (mops * st) + i)
18             end
19         end
20         if t = ws then
21             store(addr + offset)
22         end
23         offset ← (offset + st)  mod  fp;
24         i ← i + 1;
25         j ← 0; while j < cops do
26             j ← j + 1;
27         end
28     end
```

Algorithm 2: Different synthetic benchmarks configurations used for the evaluation. Assumes fixed values for $mops$, $cops$, st and fp.

```
1
2  Function CpuR(src, reps)
       Input: src: read base address
3      for i ← reps do
4          SYNTH_BENCH(src, r, mops, cops, st, fp)
5      end
6  Function CpuW Miss(dst, reps)
       Input: dst: write base address
7      for i ← reps do
8          SYNTH_BENCH(dst, ws, mops, cops, st, fp)
9      end
10 Function CpuW(dst, reps)
       Input: dst: write base address
11     for i ← reps do
12         SYNTH_BENCH(dst, w, mops, cops, st, fp)
13     end
14 Function CpuRW(dst, reps)
       Input: dst: write base address
15     for i ← reps do
16         SYNTH_BENCH(dst, rw, mops, cops, st, fp)
17     end
```

When trying to mitigate memory interference, it is common to check the validity of the work produced against the worst-case memory interference that can be experienced on a system. Many works in the state of the art [30, 14, 22, 16, 15, 31] have operated under the assumption that the worst-case access can be modeled by: (i) choosing $t = r$ or $t = ws$ (i.e., instantiating *CpuR* or *CpuW 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 Chapter 4 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. This can be problematic: a wrong worst-case estimate means that an interference mitigation technique may not work under certain circumstances.

2.5.2 ***PartitionedChaser*** synthetic benchmark

While the synthetic benchmarks introduced in Section 2.5.1 can be used to explore the effects of main memory interference and cache evictions, they cannot be used to isolate the effects of cache *congestion*, due to the inability to prevent tasks from evicting the data of one another from shared caches. For this reason, we design another synthetic benchmark – *PartitionedChaser* – to measure the slowdown from cache interference with and without eviction from the shared cache. This allows us to isolate *line conflict contention* and *congestion* slowdowns when multiple cores concurrently use the shared cache.

To produce cache accesses, the main loop of *PartitionedChaser* (the part which we measure) is a simple *pointer chasing* function, which exclusively executes reads across a circular buffer in a loop – a common way to generate cache accesses [4]. The function (POINTER_CHASER) is presented

in Algorithm 3. Every *load* (memory read) depends on the result of the previous one, meaning the memory operations are executed sequentially, in-order. How the cache is accessed depends entirely on which physical memory addresses are read from the circular buffer.

Algorithm 3: Synthetic Benchmark - *PartitionedChaser*

```
1 Function POINTER_CHASER(start, iter)
      Input: start: starting address of the circular buffer, iter:
             number of time to repeat the read of the entire circular
             buffer
2     curr ← *start;
3     while iter > 0 do
4         while curr! = start do
5             curr ← *curr;
6         end
7         iter ← iter − 1;
8     end
```

PartitionedChaser employs explicit and configurable software set-based cache partitioning, which can be used on all the platforms we previously introduced. Allowing cores to access the same partition enables line conflicts and thus evictions, allowing us to study the effects of cache *conflict contention*. Forcing cores to only access distinct partitions disables eviction, allowing to study the effects of cache *congestion*.

With N compute units executing *PartitionedChaser*, each should have access to at most $1/Nth$ of the cache. To do so, we limit a core to using $1/Nth$ of the *sets* inside each *way*. This ensures cores assigned different partitions are unable to evict the data of one another, even if they are using the same *ways*. Which *set* is used depends on the physical memory address of the data accessed (while which *way* is targeted cannot be controlled from software).

Regular buffer allocations under Linux happen across multiple pages, which start at entirely disjoint physical memory addresses. This normally makes it impossible to control which *set* is accessed in userspace. To address this, the program first obtains a portion of physically contiguous memory

from the Linux kernel to use for the circular buffer. This is done by using Linux' Contiguous Memory Allocator (CMA), with a Linux kernel compiled with `CONFIG_STRICT_DEVMEM` disabled, which allows programs to access and map physical memory. A different portion of the allocated memory is assigned to each core running *PartitionedChaser*, such that they do not use addresses which could cause coherence traffic.

To do cache partitioning on the obtained contiguous memory, we logically divide it to reflect the structure of ways and sets inside of the cache. This is to control which sets should be targeted by the accesses executed by $PartitionedChaser$.

First, we divide the contiguous memory into *Slices*, which mirror cache ways at a high level. Each *Slice* is a block of physically contiguous memory, of size $SliceSize$. A property of this setup is that accesses done at the same position inside of two different *Slices* target the same sets inside of said *Slices*. Then, we specify a *Partition*, which is the portion of each *Slice* that a core is allowed to use, of size $PartitionSize$. This effectively implements the partitioning, as each core can only access sets inside of the assigned *Partition*. Finally, the parameter *Repeat* specifies how many *Slices* should be created inside of the contiguous memory. Figure 2.6 is an illustration to help visualize how the physically contiguous memory is divided by the parameters.

$SliceSize$, $PartitionSize$ and $Repeat$ values are subject to constraints based on the target shared cache. $SliceSize$ should be smaller or equal than the size of a way in the target cache ($SharedCacheWaySize$). If $SliceSize$ exceeds this limit, accesses to different sets inside of the *Slice* may end up targeting different ways of the shared cache. Another constraint is $SliceSize >= PrivateCacheWaySize$. If that is not satisfied, some accesses will not generate an eviction from the private cache in some configurations, which is required to generate interference in the shared cache.

Assuming N cores using the shared cache, $PartitionSize$ should be $<=$ $SliceSize/N$, otherwise, there would not be enough space for N cores to use the cache at the same time without evicting the data of each other. Of note is that due to the effect of hardware prefetchers, it may not be possible to have $PartitionSize = SliceSize/N$ on some platforms, as the cores may

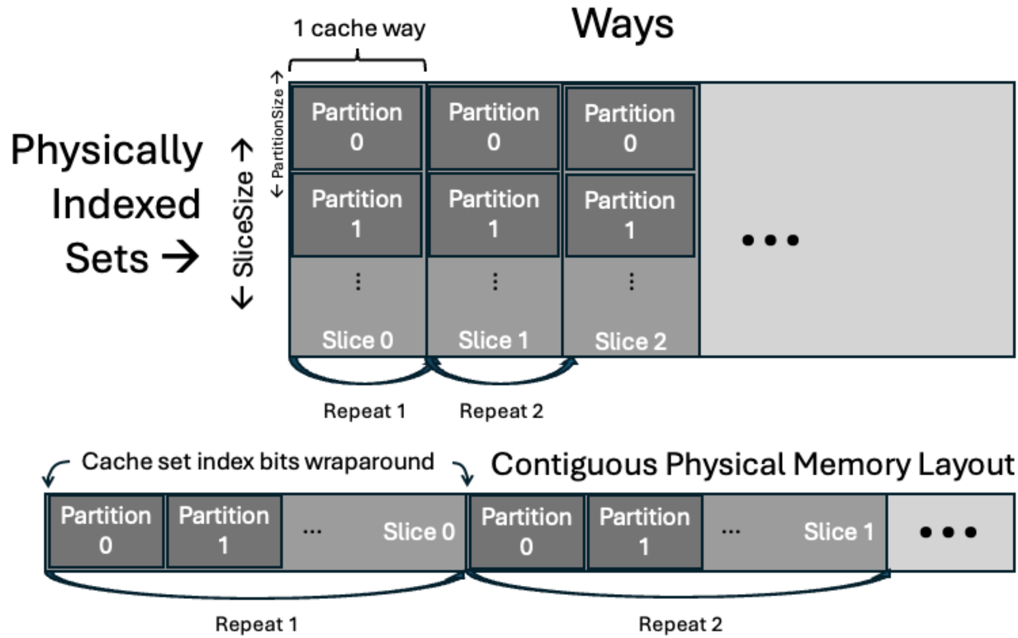


Figure 2.6: Logical division of the physically contiguous memory implemented by the cache partitioning of *PartitionedChaser*.

otherwise cause evictions to the other cores if the prefetcher exceeds the partition boundary.

Finally, *Repeat* must be chosen such to avoid exceeding the capacity of the target shared cache and executing accesses inside of lower memory hierarchy levels. As an example, when choosing $SliceSize = SharedCacheWaySize$, *Repeat* should not exceed the number of ways in the shared cache (i.e. 16 for all our reference platforms). Due to the cache replacement policy, it may not be possible to make use of all the ways in the cache without there being unwanted self-eviction from the core. As such, care must be taken in choosing a value of *Repeat* for which such phenomenon doesn't happen.

With this logical overlay applied to the contiguous memory, and the parameter constraints specified, we can implement software set-based cache partitioning in *PartitionedChaser*.

The BUFFER_POPULATOR function (Algorithm 4) populates the circular buffer using the cache partitioning scheme described above. In every *Partition*, the pointers get populated so that they force one cache access af-

ter the other (by being spaced by the size of a cache line, innermost loop). Then, the final pointer at the end of each *Partition* is changed to point at the beginning of the *Partition* in the next *Slice* (outer loop). To complete the circular buffer, the last pointer of the *Repeat Partition* is overwritten to point to the beginning of the *Partition* in the first *Slice*. With the circular buffer populated, the *pointer chasing* can access the shared cache while maintaining the partitioning.

Algorithm 4: *PartitionedChaser* Buffer Population

```
1  Function BUFFER_POPULATOR(addr, SliceSize, PartitionSize,
     Repeat)
       Input: addr: starting address of the physically contiguous
              memory, cache_line_size: size of the cache lines for the
              target platform, SliceSize: size of the Slices in which the
              memory is divided into, PartitionSize: size of the
              partition of each slice that PartitionedChaser is allowed
              to use, Repeat: amount of Slices that the circular buffer
              spans
2      iter ← 0;
3      slice_addr ← NULL;
4      while iter < Repeat do
5          slice_addr ← addr + (iter × SliceSize);
6          target_addr ←
            slice_addr + PartitionSize − cache_line_size;
7          while slice_addr != target_addr do
8              *slice_addr ← slice_addr + cache_line_size;
9              slice_addr ← *slice_addr;
10         end
11         iter ← iter + 1;
12         *slice_addr ← addr + (iter × SliceSize);
13     end
14     *slice_addr ← addr
```

2.5.3 The Polybench-ACC benchmark suite

Synthetic benchmarks enable precise control of the generated memory traffic and, presumably, interference. However, the memory access patterns gener-

ated 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 thesis, the Polybench-ACC benchmark suite [50]. 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. Each benchmark comes with a set of predefined sizes for the input dataset that an user can choose from, depending on the goal of the test. 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 the `correlation` benchmark we changed the default setting to STANDARD_DATASET, as the default (LARGE_DATASET) would cause the benchmark to take significantly longer to execute. For `convolution-3d` and `convolution-2d`, LARGE_DATASET is the default setting, and we did not alter it.

Chapter 3

Related work

In this Chapter, we describe how the issue of memory interference has been tackled in the literature, and how it relates to this thesis.

3.1 Memory interference exploration

In general, there are many papers that explore memory interference in the context of MSoCs and (more recently) HeSoCs [1, 2, 13, 51, 16, 52, 14, 15, 53, 54, 55, 56]. Typically, they fall into one of three main categories: platform-specific memory characterization, DRAM maximum interference estimate, and cache interference characterization.

3.1.1 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 HeSoCs and GPGPU-based HeSoCs).

Bansal et al. [13] propose a deep characterization of the memory systems present on the *Xilinx ZU9EG*, with some focus on the interference which the CPU cores can cause to each other. Manev et al. [51] tried to analyze the memory performance and requirements of accelerators on the *Xilinx ZU9EG* and the *Xilinx ZU3EG*. Capodieci et al. [16] highlight how on *Nvidia*'s plat-

forms, instead, DRAM performance has evolved over the years, with the rapid increase in GPU performance.

In Chapter 4, we also performed extensive tests on different embedded HeSoCs from different vendors and with different features to characterize various possible interference effects.

3.1.2 DRAM maximum interference estimate

Memory interference in MSoC has been a significant research focus since these systems were introduced. Several studies have focused on estimating the Worst-Case Execution Time (WCET) using various analytical methods. Andreozzi et al. [57] employed mixed integer linear programming (MILP) to provide a rigorous estimation. Other works have also utilized synthetic loads. Radojkovi et al. [14] focused on singular load types. Nowotsch et al. [15] instead combined different types of memory access patterns, such as *CpuW Miss* and *CpuR*. Hyoseung et al. [58] 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 [59, 10, 24, 27, 30].

Together, these approaches represent a range of strategies for ensuring more predictable performance in MSoC.

3.1.3 Cache interference characterization

Cache interference can have significant effects on execution times (reaching slowdowns of up to 346× [12]). For this reason, cache interference has been significantly studied throughout the years.

When it comes to cache *line conflicts* [1, 2] (the most known type of cache interference), Radojkovic et al. document slowdowns of up to 14× on tasks co-scheduled on an Intel Atom processor [14]. H. Yun et al. further looked into cache eviction-based slowdown and how cache partitioning may reduce it [4]. Using synthetic benchmarks, they found that cache interference from

out-of-order memory operations can cause slowdowns of up to 104×. While applying cache partitioning does reduce the slowdowns they see, a significant portion remains (up to 14×). They conclude that the slowdown under cache partitioning was due to contention of the shared MSHR component when cores are accessing data in the DRAM. They do notice some slowdown even when no accesses are done to the DRAM (i.e. due to the cache congestion we present in this thesis). However, they attribute the slowdown they observe to out-of-order memory operations overutilizing internal structures of the shared cache, and they state that when it comes to in-order memory operations (like the pointer chasing that our synthetic benchmark does), cache partitioning should remove almost all of the interference experienced by other tasks. Section 4.3 challenges this notion, experimentally proving that even in-order memory operations may cause cache congestion (causing slowdowns of up to 3×), while also showing how this phenomenon can have an effect on real-life workloads. Later, Valsan, H. Yun et al. [5] demonstrated slowdowns of up to 21× due to sharing of cache structures related to DRAM, even if cache partitioning is active. They attribute the slowdown to MSHR contention by the *interference generator* cores. They then propose a hardware and software collaborative scheme in the paper to reduce MSHR contention, which does significantly reduce cache *congestion* slowdowns as well. Bechtel and H. Yun [12] explore how synthetic benchmarks may cause slowdowns of up to 346× when caches are partitioned. It exemplifies the effects of contention on both the MSHR and the WriteBack Buffer, which both completely lock the cache when filled. We observe the exact same type of interference in Section 4.2.3. They propose bandwidth throttling to reduce the cache interference, which is the solution we end up using in Chapter 5.

3.2 Memory interference mitigation

In the literature and in this thesis, memory interference exploration is a first step towards memory interference mitigation. Memory interference mitigation has been a focus of many papers, and this section highlights how our solution (Chapter 5), with its focus on HeSoCs, compares to the literature.

A common approach for memory interference mitigation is to force mutual exclusive access of the shared main memory [10, 60, 23, 61]. The benefit of such an approach is that it provides timing guarantees, regardless of the workload being executed. The drawback is that forcing mutual exclusive accesses has been proven to lead to severe memory under-utilization [22]. *CBM*, instead, provides timing guarantees and keeps memory utilization high.
Another kind of approach present in the SoA is WCET-profiling-based task scheduling and bandwidth allocation [62, 63]. While this kind of interference mitigation does also lead to higher memory utilization, the problem is in the profiling needed to obtain the WCET of the tasks. Knowing which tasks are going to be executed and how they behave is required beforehand, meaning such a model is not particularly suited for a generic environment. The policy presented in this thesis does have an initial static setup phase, but it is adaptive in general, and thus can work in more scenarios, making it more generic.
Other works use dedicated hardware components to do monitoring and throttling to keep memory utilization high [29, 25]. The problem is that even if these implementations are evaluated on HeSoCs, they only focus on interference mitigation among CPU cores, ignoring the significant amount of interference that accelerators can cause to other actors.
The closest papers to the solution we present in Chapter 5 are [64, 17]. Zini et al. [64] propose a way to mitigate memory contention caused by external I/O components. They use the ARM QOS-400 to do bandwidth regulation, and properly study the effect of the interference mitigation approach they implement on the I/O devices. Serrano-Cases et al. [17] combine various hardware QoS mechanisms present on the *Xilinx Zynq Ultrascale+* (ARM QOS-400 and its extensions, as well as a DDR-controller) to reduce memory interference slowdown to an acceptable maximum. The model they use has various tasks execute concurrently, on both the CPU and the FPGA. The work of both Zini et al. and Serrano-Cases et al. focuses on studying the ability of the QoS mechanisms used to interact with one another. In particular, the QoS is only ever statically assigned during experiments. Section 5, instead, provides a policy that does bandwidth regulation, able to automat-

ically throttle the bandwidth by the correct amount to reach a QoS target, and to dynamically adapt to changes in the system bandwidth utilization.

Chapter 4

Memory interference exploration

In this chapter, we explore memory interference on various HeSoCs to expose the amount of slowdown that CPU cores can cause each other. Memory interference exploration of a platform is a necessary step to develop memory interference mitigation techniques. It allows to understand how different workloads may be affected by the different types of interference, and which are the problematic cases for a specific HeSoC. We do this exploration first for the main memory, and then for the shared caches. For the latter, we differentiate between cache conflict contention (a phenomenon caused by spatial interference in the caches), and cache congestion (the same type of interference we observe when it comes to the main memory). We focus on CPU cores exclusively because on our reference platforms the accelerators do not share a cache with the CPUs, meaning that their contribution would only apply to main memory interference. The effects of accelerator-based main memory interference are later explored in 5.

4.1 Main memory interference

To analyze main memory interference, we measure the execution time of a *task under test* running on one of the cores, with the *interfering tasks* exe-

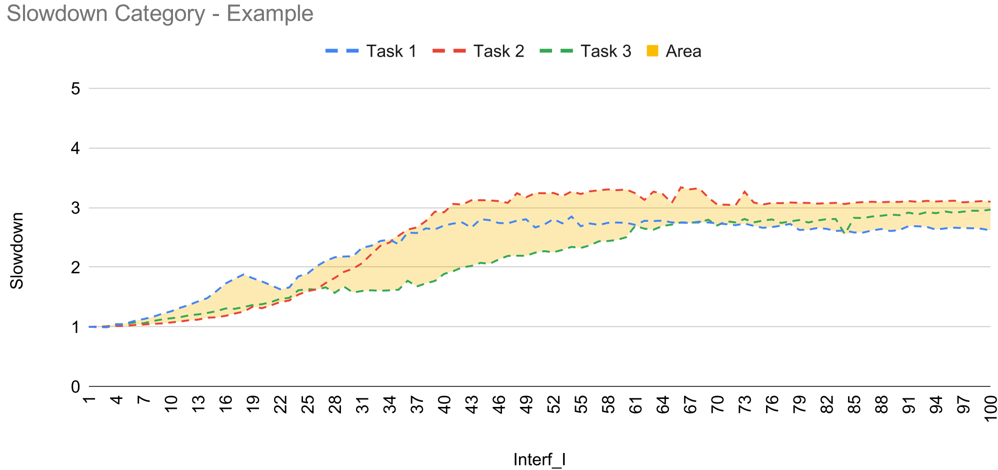


Figure 4.1: Example of how the slowdown of multiple Polybench benchmarks combines to create the area of one category in Figure 4.2.

cuting on the remaining CPU cores, and accessing the main memory. These tests are conducted on two of our reference HeSoCs: the *Xilinx ZU9EG* and the *Nvidia TX2*. We chose to conduct this analysis on these two platforms as they are respectively a FPGA-based HeSoC and a GPU-based HeSoC with comparable hardware when it comes to the CPU (both have one CPU cluster, with four cores in it).

The *task under test* is either one of our synthetic benchmarks or one of the Polybench benchmarks (see Sec. 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 the target platforms. *Interfering tasks* access memory at a controllable intensity I by changing the ratio between $Mops$ and $Cops$ (see Sec. 2.5.1). 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 the interfering task memory intensity *Interf_I*.

Figures 4.2a, 4.2c, 4.2e and 4.2b, 4.2d, 4.2f present the interference analysis results obtained using the *Nvidia TX2* and the *Xilinx ZU9EG*, respec-

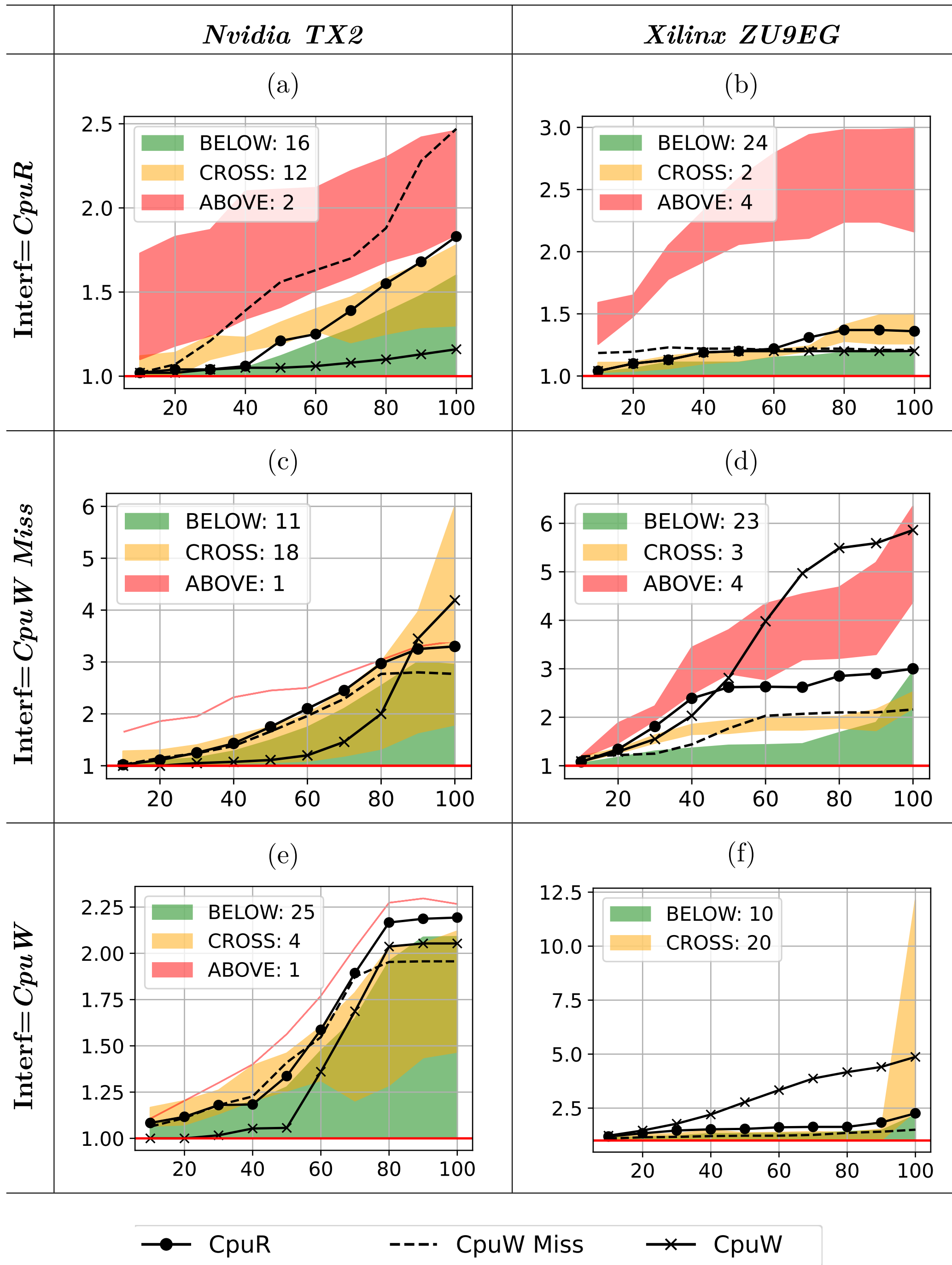


Figure 4.2: ***Nvidia TX2*** **(a,c,e),** ***Xilinx ZU9EG*** **(b,d,f)**. Time increase for Synthetic and Polybench benchmarks. On the X axis, the memory intensity percentage *Interf_I* of the *interfering tasks*. On the y axis, the slowdown that the benchmark was subject to. Both labels are not reported to save space. Note: each plot has a different maximum y axis value.

tively. There is one subplot for each interference traffic type (*CpuR*, *CpuW Miss*, *CpuW*). The experiments were run 10 times for each benchmark due to the high amount of time some of the Polybench require to run (in the order of more than 2 minutes per execution), 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 Figure 4.2. 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 legend indicating how many Polybench benchmarks fall in one specific category. The red bottom horizontal line indicates the base slowdown when executed in isolation (i.e. 1.0×). 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 *Interf_I* 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 Figure 4.1. The categories in Figure 4.2 are: i) **ABOVE**: those benchmarks that suffer more interference than the *CpuR* synthetic benchmark throughout the entire *Interf_I* range; ii) **CROSSING**: those benchmarks that suffer more interference than the *CpuR* synthetic benchmark only for some *Interf_I* configurations; iii) **BELOW**: those benchmarks that suffer less interference than the *CpuR* synthetic benchmark. *CpuR* (black curve with black circular points) was used as the divider between categories as it was assumed by previous works [30, 14, 22, 16] 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 Figure 4.2**. **However, one can quickly notice that is not the case**, as highlighted by the number of benchmarks falling into either categories (reported in the legends of Figure 4.2). In general, it can be observed that the *CpuR* 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 4.2a and 4.2b present the slowdown perceived by real and synthetic benchmarks when running in the presence of *CpuR interfering tasks*, an interference configuration which some literature [14, 22, 16] considered to be the worst-case one. Subplot 4.2a 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 4.2b 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 4.2c and 4.2e show that the worst-case interference on *Nvidia TX2* is generated by *CpuW Miss* (Subplot 4.2c), which causes R/W traffic due to the cache-line update operation. This result is in line with how other literature [30, 15, 31] conducts interference analysis. However, subplots 4.2d and 4.2f show that on the *Xilinx ZU9EG* hardware the worst-case interference traffic type is *CpuW* (Subplot 4.2f), which causes slowdowns of up to $\approx 12\times$. This is a type of interference which is not often investigated, even among literature which is more thorough and checks for *CpuW Miss* traffic.

4.1.1 Analysis Summary

The results presented in this section demonstrate that on the *Xilinx ZU9EG* and the *Nvidia TX2*, *CpuR* is neither the workload causing the highest amount of interference (highlighted by subplots 4.2c and 4.2f) nor the one most sensitive to memory interference (as seen in all subplots of Figure 4.2). In fact, the results are highly platform-dependent, with the two analyzed MSoCs having different worst-case *interfering tasks* (*CpuW Miss* on the *Nvidia TX2* and *CpuW* on the *Xilinx ZU9EG*). Figure 4.2, in general, highlights the need to do proper memory interference exploration, 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. This may seem counterintuitive, since the synthetic benchmarks are supposed to be the tasks subject to the worst-case interference. The reason for this will be further explored in Section 4.2.

4.1.2 Comparison with the literature

There are some examples of underestimating the worst-case in the literature that we already cited in Chapter 3. In this subsection, we highlight how the results we present in this section can be used to improve memory interference characterization in the literature.

Capodieci et al. [16] measure the effect of memory interference on *Nvidia*'s platforms (including on *Nvidia TX2*). In their experiments they use *CpuR* to measure interference effects as the *task under test*, which was proven as the task not most subject to interference in this section. This means that the worst-case when it comes to DRAM-based interference is underestimated. Cavicchioli et al. [22] 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 *CpuR* as both the *task under test* and the *interfering tasks*, which means that the worst-case is underestimated, as the results highlight. Brilli et al. [30] present a work on modern HeSoC memory regulation and control, which expands on the work from Cavicchioli et al. [22]. They evaluate CMRI on both the *Nvidia TX2* and *Xilinx ZU9EG*. The previous analysis is expanded with *CpuW Miss* as another possible *interfering task*. However, in their experiments they: i) do not take into consideration *CpuW* as a *interfering task* (important for the *Xilinx ZU9EG*, as shown in subplot 4.2f); ii) still only consider *CpuR* 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.[31] try to bound memory interference delay in COTS MSoCs. During the evaluation, they make use of the *STREAM* benchmark [65] 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 *CpuW 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 *CpuW* as the *interfering task*. Nowotsch et al. [15] in their maximum DRAM interference estimation only investigate the interference of *CpuW Miss* and

CpuR traffic, meaning that their methodology may not find the actual worst-case on some platforms, as we have demonstrated in this thesis.

Radojkovi et al. [14] present WCET bound estimates which also make use of *CpuR* 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*.

4.2 Cache conflict contention

In this section, we explore why specific real-world benchmarks (e.g. some of the Polybench) are more sensitive to memory interference than synthetic ones – which are modeled to capture worst-case behaviors. In general, when conducting memory 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 spatial interference, providing a correlation between the traffic generated by the *interfering tasks*, and their effect on the memory performance of our evaluation setup. Cache spatial interference has been addressed in the literature using techniques like *shared cache partitioning* [39, 40]. 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 *CpuR* synthetic benchmark acting as a *task under test*, under different 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., *CpuW Miss* and *CpuW*), generate a memory traffic that either pollutes the cache (i.e., *CpuW Miss*), making it difficult to analyze the cache effects, or does not access the cache memory at all (i.e., *CpuW*, see Section 2). During each test run, we keep track of the LLC refills using Performance Monitoring Units (PMUs) [66]. This is because the LLC refills of the *task under test* increase when it tries to read data that was in the

shared cache and that has been evicted by the *interfering tasks*. Since the two hardware platforms that we use have different memory configurations (i.e., 1MB and 2MB 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 512KB, 768KB, 2048KB, and 16384KB. For the *Nvidia TX2*, we use 1024KB, 1536KB, 4096KB, and 32768KB 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 behaviour in the results. The interference is generated using our synthetic benchmarks – one for each remaining core – configured with a fixed *fp* of 16MB, 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 *Interf_I* parameter (see Section 4.1) 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 *Interf_I* (in general, the results were within 5% of the average).

Figures 4.3 and 4.4 present the results of the experiments for the *Nvidia TX2* and the *Xilinx ZU9EG*, respectively. In particular, Figures 4.3a, 4.3b, 4.3c and 4.4a, 4.4b, 4.4c present the **slowdown** of the *task under test*, while 4.3d, 4.3e, 4.3f and 4.4d, 4.4e, 4.4f 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., *CpuR*, *CpuW Miss* and *CpuW*) and the *Interf_I* parameter configuration. In the following subsections, we discuss the results for each configuration of the *interfering tasks*.

4.2.1 *CpuR* interfered by *CpuR* - Refills

Subplots 4.3a, 4.3d, and 4.4a, 4.4d show the results obtained by using *CpuR* as *interfering tasks*. As expected, the results heavily depend on the *fp* con-

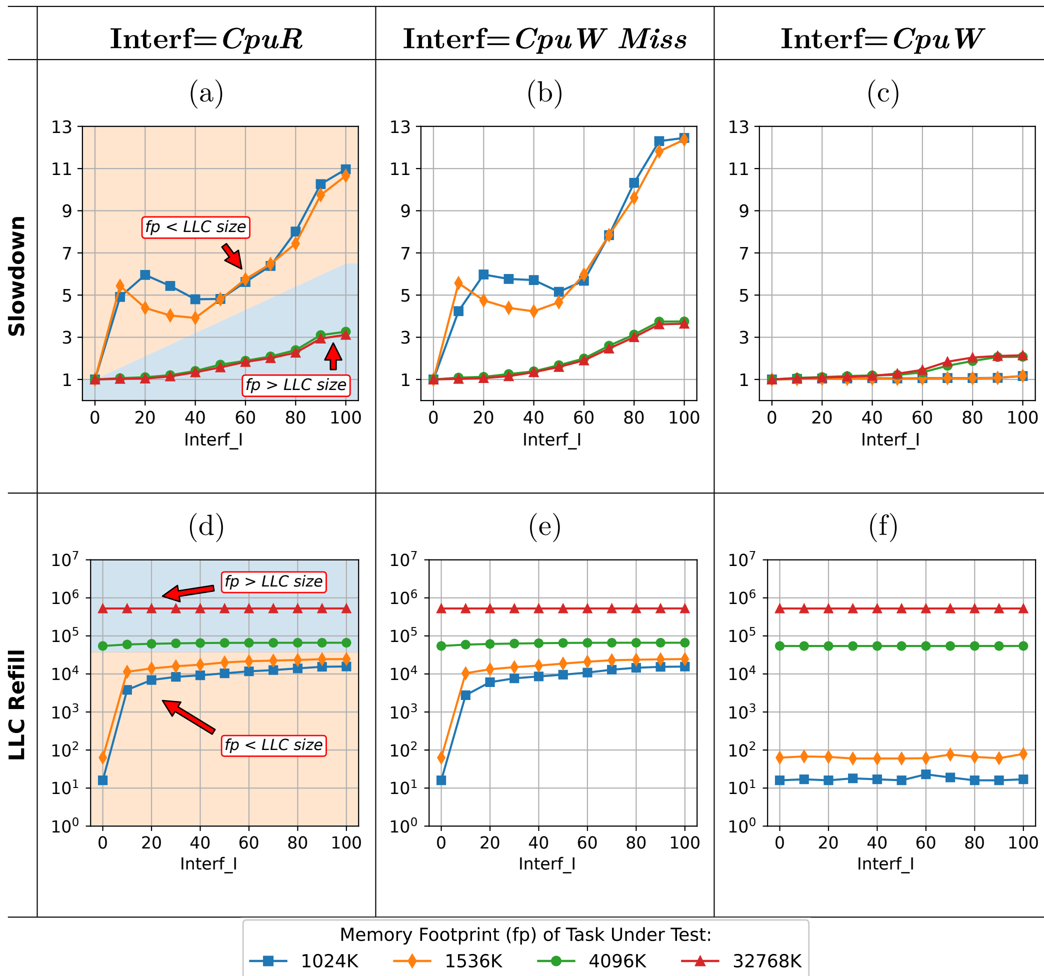


Figure 4.3: ***Nvidia TX2.*** Slowdown and LLC refills triggered by the *task under test* as a function of the *interfering tasks* configuration.

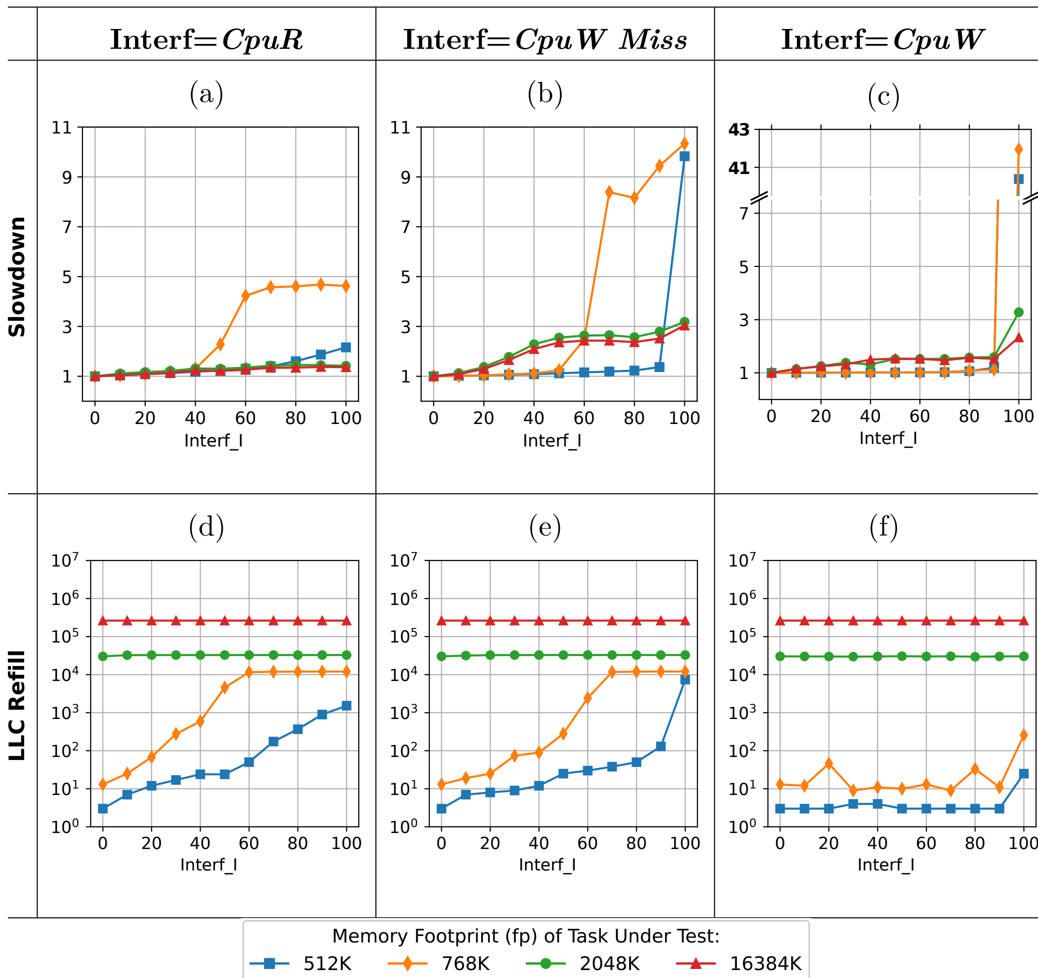


Figure 4.4: ***Xilinx ZU9EG.*** Slowdown and LLC refills triggered by the *task under test* as a function of the *interfering tasks* configuration.

figuration 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 *Interf_I* setting used for the *interfering tasks*. This is highlighted in Figure 4.3d for *fp* configurations of 4096K and 32768K, and Figure 4.4d for configurations of 2048K and 16384K. As a result, the number of LLC-to-DRAM transactions is constant, and the slowdown shown in Figures 4.3a and 4.4a 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 the *Xilinx ZU9EG*. On the other hand, when *fp* is smaller than the LLC size and the *interfering tasks* are muted (i.e., 0% *Interf_I*), 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 Figure 4.3a for *fp* configurations of 1024K and 1536K, and in Figure 4.4a for configurations of 512K and 768K. As the *Interf_I* 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 (Figure 4.3a), and 4.5× on the *Xilinx ZU9EG* (Figure 4.4a).

4.2.2 *CpuR* interfered by *CpuW Miss* - Refills and Writebacks

Figures 4.3b, 4.3e, 4.4b, and 4.4e report the slowdown and LLC refill results obtained using *CpuW Miss* as *interfering tasks*. In this configuration, the results reported in Figures 4.3e and 4.4e show that the LLC refills are comparable to the ones registered in the *CpuR interfered by CpuR* configuration (Figures 4.3d and 4.4d). However, the slowdown results (Figures 4.3b, 4.4b) 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 Figures 4.5 and 4.6. In case of *READ_MISS* interference (Figures 4.5a, 4.6a), the number of LLC write-

Nvidia TX2

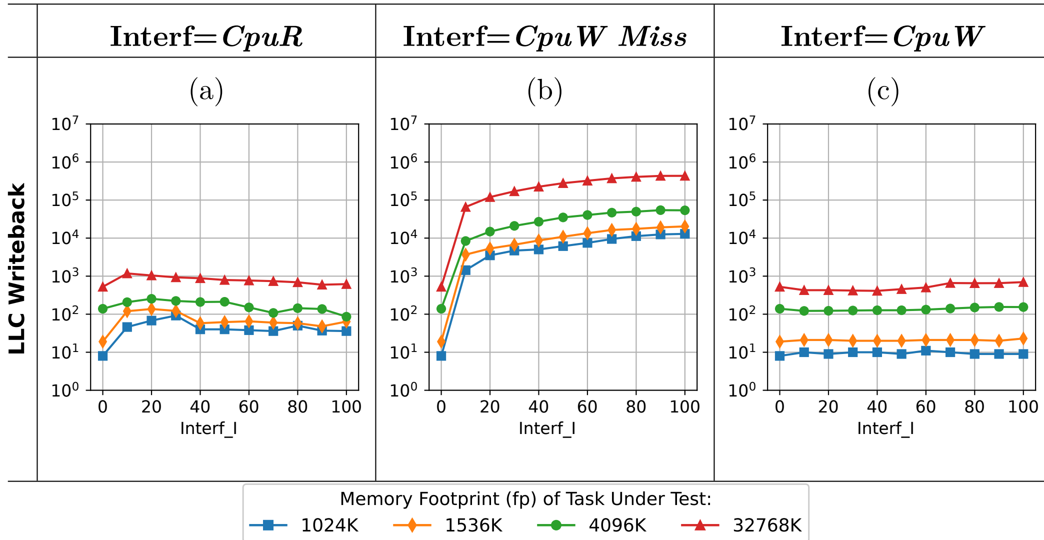


Figure 4.5: ***Nvidia TX2.*** LLC writebacks triggered by the *task under test* as a fucntion of the *interfering tasks* configuration.

Xilinx ZU9EG

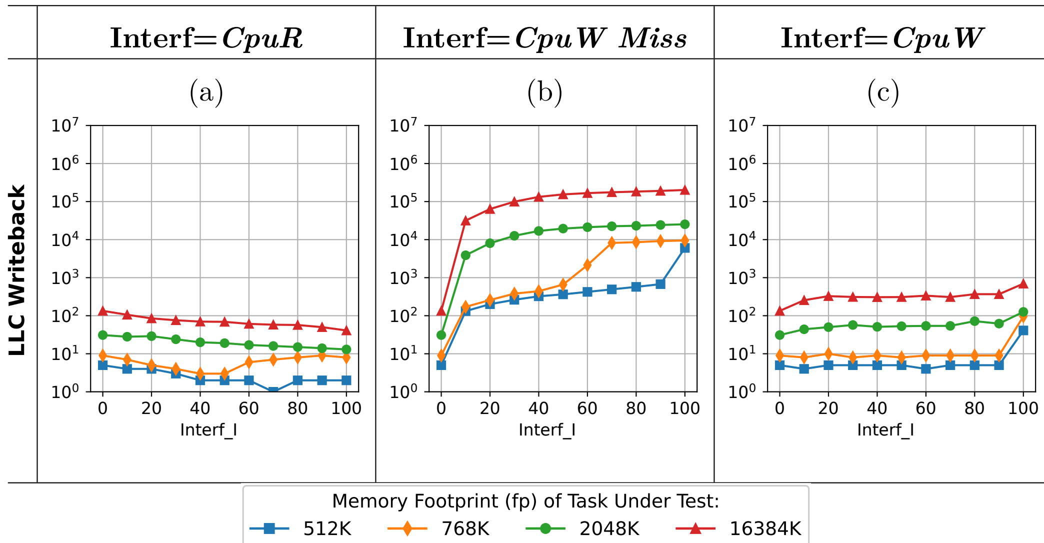


Figure 4.6: ***Xilinx ZU9EG.*** LLC writebacks triggered by the *task under test* as a fucntion of the *interfering tasks* configuration.

backs is less than 10^3 for each *fp* configuration that is used. In the same configuration, the number of LLC refills (see Figures 4.3d, 4.4d) 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 *CpuR interfered by CpuW Miss* configuration (Figures 4.5b, 4.6b) grows accordingly to the *Interf_I* 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 *CpuR interfered by CpuR* 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.

4.2.3 *CpuR* interfered by *CpuW* - Filling the write buffer

As explained in Section 2, the traffic generated by *CpuW 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 Figures 4.3c, 4.3f, 4.5c, and 4.4c, 4.4f, 4.6c demonstrate this aspect. In fact, the impact of the *Interf_I* parameter on the number of LLC refills/writebacks is almost null or negligible if compared to other configurations (see Sections 4.2.1, 4.2.2). Nevertheless, the slowdown results are very different between

CpuR* (*fp* = 512KB) interfered by *CpuW

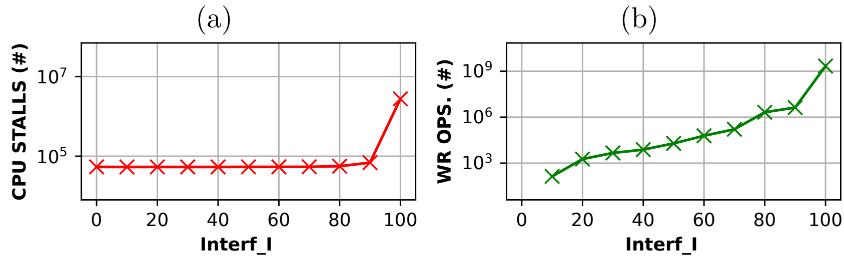


Figure 4.7: *CpuR* (*fp* = 512KB) interfered by *CpuW*. The *CPU STALLS because of load miss* (Subplot 4.7a) are measured on the *task under test*. The *WRITE OPERATIONS that stall the pipeline because the store buffer is full* (Subplot 4.7b) are measured on the *interfering tasks*

the two setups. On the *Nvidia TX2* (Figure 4.3c), 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 behaviour of the cache and therefore do not create cache space contention. On the *Xilinx ZU9EG* (Figure 4.4c), 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 *CpuW Miss* interference, and a **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 [42]. 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 [42]. 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 [12]. 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 *CpuR* interfered by *CpuW* scenario: the *write-no-allocate* traffic generated by the *CpuW interfering tasks* can saturate the store buffer, causing the LLC cache to block. This, in turn, can cause the CPU core executing the *CpuR* 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 *CpuW 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 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 Figure 4.7. In the figure, the subplots present the results as a function of the *Interf_I* parameter of the *interfering tasks*. As shown in Subplot 4.7b, when the *Interf_I* 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 4.7a) grows from $\approx 10^5$ to $\approx 10^7$. This result indicates that when the *Interf_I* parameter reaches 100%, the store buffer gets saturated by the memory requests of the *CpuW interfering tasks*, causing the CPU core executing the *task under test* to stall, and ultimately, the observed slowdown. On the *Xilinx ZU9EG*, we have experimentally observed that this stalling happens when two or more cores are generating *write-no-allocate* traffic. In general, these results, along with the findings we present in Section 4.3, suggest that the *Xilinx ZU9EG* in particular uses a cache that is not able to handle the capabilities of the CPU cores of the platform in terms of bandwidth under heavy loads.

4.2.4 Analysis Summary

This analysis allows us to draw several results:

- We observe that cache-bound benchmarks (i.e., *fp* $<$ LLC size) suffer from greater slowdown (11× on the *Nvidia TX2* and 4.5× on the *Xilinx ZU9EG*) when compared to the DRAM-bound ones (3× on the *Nvidia TX2* and 1.5× on the *Xilinx ZU9EG*, Section 4.2.1). This phenomenon demonstrates that, on the analyzed platforms, cache conflict contention is a more severe problem than DRAM interference.

- We demonstrate 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 causing even more interference, with a 12.5× slowdown reported on the *Nvidia TX2*, and a 10× slowdown on the *Xilinx ZU9EG*, see Section 4.2.2). This has proven to be a major source of slowdown on our setup, particularly on *Xilinx ZU9EG*.

- 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 (with a 42× slowdown reported on the *Xilinx ZU9EG*, see Section 4.2.3).

4.3 Cache congestion

Cache conflict contention is a type of *spatial interference* that is specific of shared caches. However, it is not the only type of shared cache interference that is possible. Next, **we isolate the effects of *cache congestion***, i.e. multiple CPU cores accessing the same shared cache, shown in Fig.4.8. When this happens, CPU cores cause each other *temporal interference*, just like in the main memory.

We run the experiments presented in this section on the *Xilinx ZU9EG*, the *Nvidia AGX Orin*, the *Raspberry Pi 4* and the *Raspberry Pi 5*. We run the experiments on the *Nvidia AGX Orin* instead of the *Nvidia TX2* as there were issues getting *PartitionedChaser* working on it. In general we use the *Nvidia AGX Orin* as it is a newer GPU-based HeSoC with the same

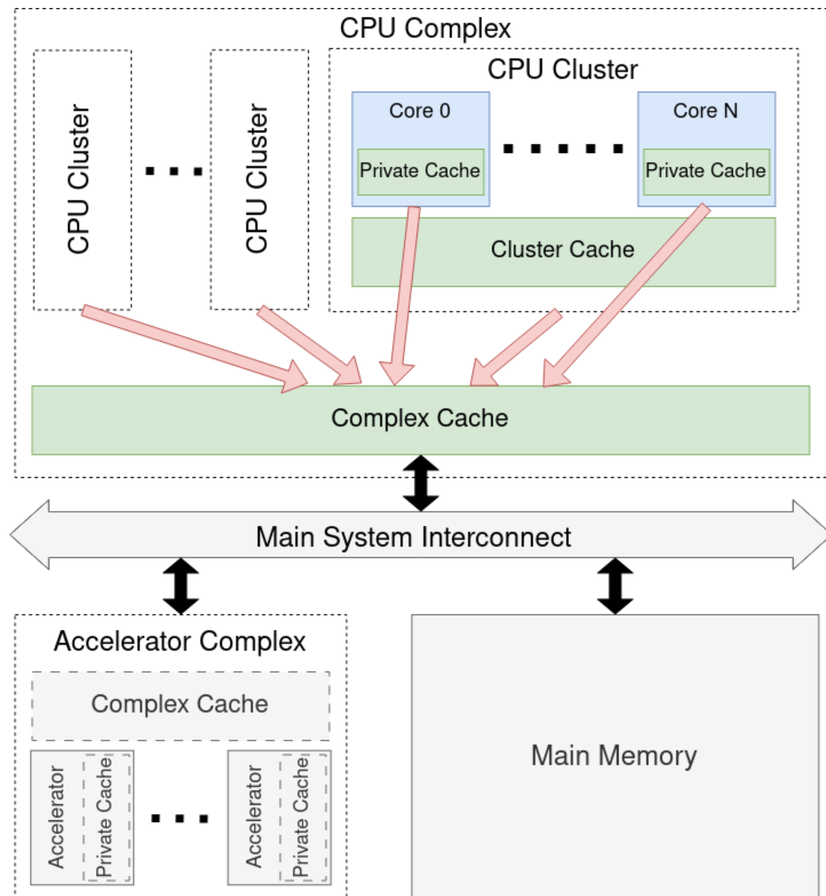


Figure 4.8: Diagram showing how cache congestion can be caused by all the cores in a generic HeSoC making use of the same cache, without the need for *spatial interference*. In gray, the unused components when it comes to cache congestion for this example. Notice how the Main Memory is not used by any of the cores (due to the absence of *spatial interference*).

	Xilinx ZU9EG	Raspberry Pi 4	Raspberry Pi 5	NVIDIA Orin AGX
SliceSize	64	32	128	128
PartitionSize	16	8	16	16
Repeat	16	16	16	9

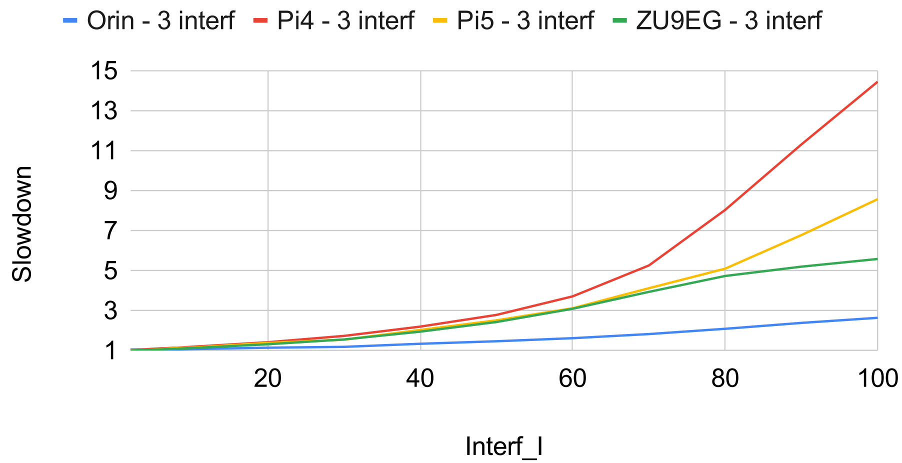
Table 4.1: Platform-specific values used for executing *PartitionedChaser*

CPU cluster layout as the *Nvidia TX2*. We use the *Raspberry Pi 4* and the *Raspberry Pi 5* (despite them being MSoC and not HeSoC) to get more data points on the effects of *cache congestion*.

We use *PartitionedChaser* (introduced in Section 2.5.2) to understand the effects of *cache congestion*.

For the experiment we use four CPU cores (the highest common amount of cores that shares a cache across all reference platforms) and four software cache partitions (one per core). The baseline to our measurements is the execution time of *PartitionedChaser* running without being subject to any cache interference. We then time the execution of *PartitionedChaser* on multiple cores for two different configurations: i) with evictions; and ii) without evictions. With these measurements, we assess the slowdown due to cache conflict contention and cache *congestion*, respectively, on various platforms. The reported slowdown is obtained by averaging the execution time of 1000 runs of the synthetic benchmark. We compare the results for *cache congestion* with the ones for cache conflict contention on the platforms we use to have a sense of how important of a slowdown *cache congestion* causes.

The values of *SliceSize*, *PartitionSize* and *Repeat* used for *PartitionedChaser* on the various platforms are determined by doing an automated parameter sweep, and reported in Table 4.1. The parameters were chosen with three objectives: *PartitionedChaser* having i) no cache misses when running alone (parameters within bounds specified in Section 2.5.2); ii) no increase in misses when executed on multiple cores and running in different partitions (partition slices sized to avoid prefetcher overruns); iii) selecting the configuration among those enabled by (i) and (ii) that present the highest amount of interference when multiple cores are using the same partition.



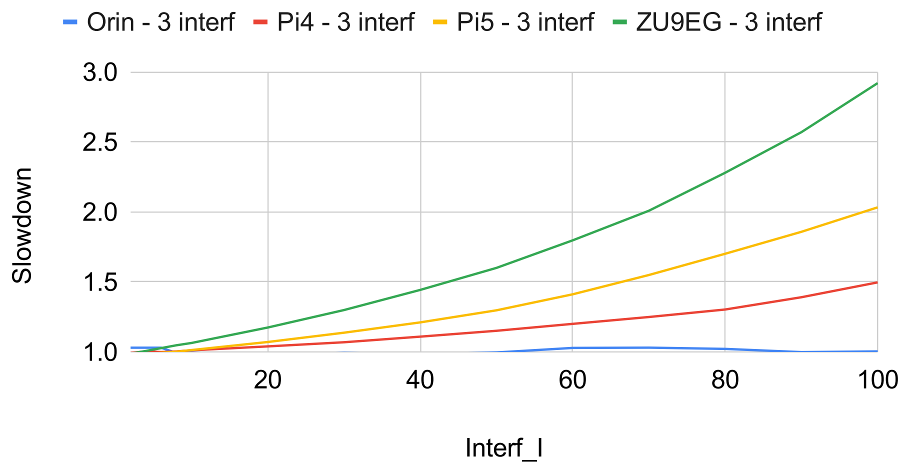


Figure 4.9: Slowdown due to line conflicts (evictions) (a) slowdown due to non-conflict contention (b).

Figure 4.9a shows the slowdown for all platforms for varying *Interf_I* when all *PartitionedChaser* instances use partition 0. This includes both the effects of *conflict contention* and *congestion*. These results clearly illustrate

how the *Raspberry Pi 4* can be especially slowed down (14×) when evictions are involved. On the *Raspberry Pi 5*, evictions can still cause up to 9× slowdown. Even the NVIDIA Orin AGX suffers significantly due to cache evictions, though it is the platform least troubled by them, with a (still significant) maximum slowdown of 3×. The slowdown curves themselves as the *Interf_I* changes are interesting as well, since they are not strictly linear. For the Raspberry Pi 4, lowering the *Interf_I* from 100 to 70 allows the slowdown to become of just 3×.

Figure 4.9b shows the slowdown for all platforms for varying *Interf_I* when all *PartitionedChaser* instances are executing on their own partitions. The slowdowns reported are exclusively an effect of *congestion*. The Xilinx ZU9EG is the platform most subject to interference, reaching a particularly high 3× slowdown at *Interf_I* 100. Interestingly, unlike in Figure 4.9a, the Raspberry Pi 5 is slowed down more due to congestion (2×) than the Raspberry Pi 4 (1.5×). Finally, the NVIDIA AGX Orin is particularly interesting as well, with it being subject to effectively no cache-congestion slowdown. It is noteworthy that of the evaluated platforms, the NVIDIA Orin AGX is the only one dimensioned not to exhibit congestion slowdowns.

In summary, Figure 4.9 proves how **even in the absence of cache misses, it is possible to observe slowdowns due to cache *congestion***. This interference cannot be mitigated by cache partitioning. Instead, we find up to an unprecedented 3× slowdown in modern COTS MSoC, differently from what is reported in the literature.

4.3.1 Polybench Cache Congestion Slowdown

We also measure the slowdown using the Polybench suite, just like for main memory interference, to confirm the congestion-based slowdowns reported in Section 4.3. As it is not possible to enforce the Polybench to use the specific cache partitioning scheme, as we used with *PartitionedChaser*, we employ the hardware-based cache partitioning of the Raspberry Pi 5 [67].

For this experiment, the *under test* core executes benchmarks from the Polybench test suite in a partition spanning half of the L3 cache, while three

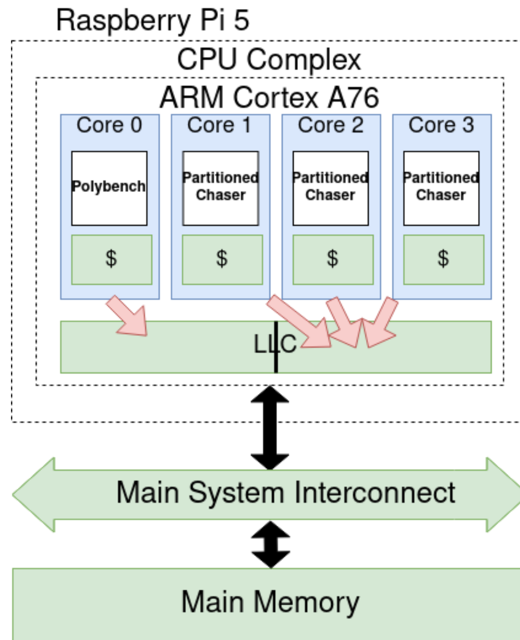


Figure 4.10: Diagram showing how the effects of cache congestion on the Polybench suite are tested on the Raspberry Pi 5.

interference generator cores execute *PartitionedChaser* in a partition spanning the other half of the L3 cache, where they are setup as to not cause data evictions to one another. This is exemplified in Fig.4.10. Due to this, only 1 of the *interference generator* cores is ever executing cache access to a particular set of the cache. This removes extra DRAM traffic as a possible source of slowdown. This is done for multiple levels of *Interf_I*, to measure how memory intensity affects slowdown for the Polybench suite. The results are compared to the baseline execution times of the Polybench suite running in isolation, with the cache partitioning scheme still applied.

Table 4.2 shows the slowdown of the 9 most interfered Polybench workloads as the *Interf_I* changes. This is compared to the slowdown *PartitionedChaser* was subject to (bottom row). The reported slowdown for the Polybench is obtained by averaging the execution time of 10 runs, just like for the main memory interference. Of note is that all presented benchmarks suffer from of cache congestion slowdowns. Most notably of all, *correlation*, which reaches a 1.82× slowdown due to cache interference from in-order mem-

Interf_I	20%	40%	60%	80%	100%
Correlation	1.05	1.21	1.38	1.60	**1.87**
Covariance	1.03	1.06	1.09	1.15	1.19
Durbin	1.00	1.00	1.00	1.02	1.16
Fdtd-2d	1.02	1.05	1.08	1.12	1.15
Floyd-Warshall	1.05	1.13	1.17	1.22	1.24
Gemver	1.03	1.07	1.10	1.13	1.18
Gramschmidt	1.00	1.00	1.00	1.05	1.18
Lu	1.03	1.08	1.12	1.16	1.20
Ludcmp	1.01	1.03	1.07	1.10	1.13
PartitionedChaser	1.08	1.22	1.41	1.83	2.40

Table 4.2: Slowdown with hardware-based cache partitioning active for Polybench on Raspberry Pi 5. Reported below the slowdown values for *PartitionedChaser* for comparison.

ory operations, which is significant. This proves that, even if no workload from the Polybench experiences the same peak slowdown as our synthetic benchmark, congestion-related slowdown is a real problem.

4.3.2 Comparison with the literature

Cache interference is widely studied, with many techniques to reduce its effects [4, 5, 6, 7, 10, 11]. The work presented in this section shows an underexplored aspect of cache interference, which we call in this thesis cache *congestion* [4]. On certain platforms, this phenomenon is observable even when no DRAM accesses are being executed, and even when executing in-order memory operations. Even if all cores are getting hits in the shared cache, tasks may still be subject to up to 3× slowdowns. In this subsection, we highlight how certain cache interference mitigation techniques may struggle against cache congestion.

Pellizzoni et al. [10] propose the Predictable Execution Model (PREM) to permit predictable task execution via a set of compilation rules [55]. Particular emphasis is put in monitoring and preventing cache misses, as they are known to be subject to interference. However, we demonstrate that cache *congestion* slowdowns happen even with cache hits, exposing another type of

cache interference which should be monitored.

Dugo et al. [7] use cache locking and partitioning to lower non-determinism and contention in lower-level caches, reducing cache interference and unpredictability. However, concurrent usage of the cache is still possible, hence the effect of cache *congestion* could still be observed.

Other researchers focus on analyzing cache usage fairness across multiple factors. Seongbeom et al. [11] propose a scheme which proves that fair cache usage directly correlates with reduced execution time, providing a different perspective on managing cache resources effectively. However, the results presented in this section show that even when cache usage is perfectly fair (as all the cores execute *PartitionedChaser* with the same configuration), severe cache *congestion* slowdowns still occur.

For cache partitioning techniques, Sparsh realized a survey for multicore processors [6]. The survey provides definitions for cache partitioning, and catalogues the existing techniques found in the literature and in the hardware according to various parameters. After that, it focuses on how cache partitioning schemes interact with other components (i.e. hardware prefetchers), as well as the possible issues that may arise from such interactions. This thesis complements the survey, as the findings about cache congestion presented in this thesis are not covered in [6].

Chapter 5

Characterization-based Bandwidth Manager

Chapter 4 exposes how much memory interference can be a problem in modern HeSoCs.

Slowdown like these may not be acceptable if the tasks that are interfered constitute part of time-sensitive applications. Such applications typically pose Quality of Service (QoS) constraints, usually expressed in terms of guaranteed timing behavior (e.g., maximum tolerated delay/slowdown). This problem has been studied extensively (as seen in Sec. 3), and several solutions have been proposed. Some approaches aim at eliminating interference by design, by imposing actors to access main memory one at a time [10, 27], in a controlled order. This approach might be overly conservative, and lead to severe underutilization of main-memory bandwidth [22]. An optimal solution must not only keep interference under control, but also avoid underutilization of the main memory bandwidth, and of the HeSoC in general.

In this respect, a more flexible, and widely adopted solution is bandwidth regulation [30, 29, 25, 18, 19, 20, 21, 26]. Interference is controlled by setting a limit to the memory bandwidth that each actor can demand, and by enforcing that limit through memory access *throttling*. In this respect, HeSoCs widened the scope of bandwidth regulation, and introduced the need for heterogeneous regulation between CPU(s) and accelerators [17].

Bandwidth regulation does enable a higher utilization of memory bandwidth to be reached, provided that bandwidth limits are properly assigned. Given the overall memory traffic generated by all actors, the set of bandwidth limits must not only be correct – i.e., such that the QoS requirements of all actors are met – but also efficient in terms of resource utilization. In this context, specifically, such that the total memory bandwidth utilization from all the actors is as high as possible while still meeting the QoS constraints.

Computing correct and optimal limits in a HeSoC is not an easy task. The bandwidth with which each actor can access main memory varies non-linearly and significantly with the intensities, the types and the spatial patterns of the memory accesses performed by all actors. Consequently, it is impossible to have simple general formulas that yield optimal configurations of limits for any possible mix of memory traffic.

Things get even more challenging if we consider that bandwidth demands and memory access patterns typically vary over time in real systems. After a change in bandwidth demands, previously computed bandwidth regulation parameters may become incorrect and/or suboptimal, thus requiring dynamic monitoring and adaptation to such varying operating conditions.

No complete solution to this complex, multi-faceted problem, is currently available in the literature. In this thesis we take a first step in filling this gap. We propose and evaluate *Characterization-based Bandwidth Manager* (*CBM*), an initial closed-loop-control bandwidth manager that regulates the bandwidth of main-memory accesses of accelerators, in the presence of a dynamic memory traffic and under the general assumption of not knowing either the total execution times of tasks, or the fractions of execution time that tasks devote to memory accesses. Therefore *CBM* does not know exactly how much a given reduction of the main-memory bandwidth for a task slows down the execution of that task.

As for the memory traffic to control, in this chapter we restrict to the simplest case that triggers the need for heterogeneous bandwidth regulation: a single *critical task* running to completion on (one of the cores of) a CPU, and performing accesses to main memory; in parallel with this task, accelerators generate an intense, but not critical, memory traffic. The QoS requirement

of the critical task is expressed as *the maximum slowdown that the task tolerates* for its execution. In this thesis, we do not consider a general scenario with multiple critical tasks and differentiated QoS requirements, because this would entail also a combinatorial explosion of the number of configurations of bandwidth limits to consider.

CBM regulates the bandwidth of only the accelerators, by computing and enforcing a global *throttling factor* (*TF*, formal definition in Section 5.1.2). It updates this factor periodically, as a function of the main-memory traffic. In particular, *CBM* computes an optimal *TF*, i.e., one that maximizes the total utilization of the main-memory bandwidth, while still guaranteeing the critical task to complete with a slowdown not higher than the maximum tolerated one. Interestingly, *CBM* manages to compute this optimal *TF* even without knowing the exact relation between offered memory bandwidth and slowdown suffered by the critical task. Finally, this factor is of course optimal within the precision allowed by the update frequency and the number of iterations needed by the control loop to converge after each memory-traffic change (typically in the range of 900 μs).

CBM keeps the above number of iterations very low, through the strategy that motivates its name. In an offline phase, *CBM* performs a characterization of how a generic CPU task may be slowed down on the platform at hand, as a function of the memory traffic generated by the task itself and by the accelerators. Then, the online closed-loop control of *CBM* uses this characterization to guess a candidate throttling factor, in one step, every time the memory-traffic mix changes significantly. In the best case, this factor happens to be correct and optimal. In the worst case, it is however close to optimal, so only minor changes, and few iterations, are needed to converge to an optimal value.

In the next sections, we first present the target platform and the system model on which *CBM* is based. After that, we introduce in detail *CBM*. Finally, we evaluate the performance of *CBM* for the system model it is based on, highlighting both its strengths and its weaknesses.

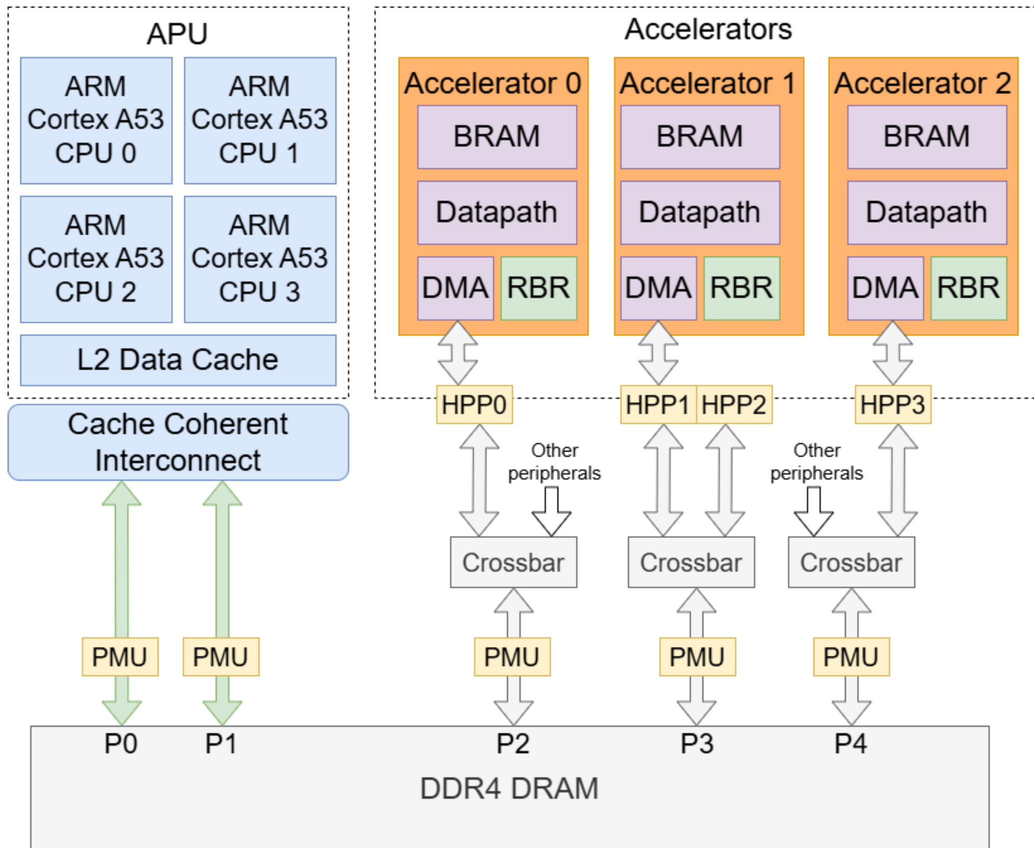


Figure 5.1: APU and Accelerators of the *Xilinx ZU9EG*, an FPGA-based shared memory heterogeneous SoC.

5.1 Target Platform and System Model

In this section, we introduce in more detail the hardware platform on which we have developed and tested our solution (the *Xilinx ZU9EG*), as well as our system model.

5.1.1 Hardware platform

Fig.5.1 shows a detailed block diagram of the *Xilinx ZU9EG* SoC, expanding on the one presented in Section 2.4.1. The diagram shows that on the *Xilinx ZU9EG*, the main memory is connected to the *CPU complex* and the *accelerator complex* via ports P0 to P4. The memory interconnect system of

the *Xilinx ZU9EG* is comprised of a Cache Coherent Interconnect (connecting the L2 Data Cache to P0 and P1) and various Crossbars multiplexing requests coming from the accelerator complex and other system peripherals. When it comes to the accelerators, on the *Xilinx ZU9EG* the FPGA logic is equipped with 4 outgoing high performance ports (HPP0 to HPP3), but two of these are multiplexed on the same external crossbar. For this reason we only consider up to three accelerators in our experimental setup, as adding a fourth one would not increase the bandwidth usage from the accelerator complex. The accelerator complex is equipped with 4MB of Block RAM [68] a fast, explicitly managed memory to store data locally, distributed across the various accelerators. Each accelerator is equipped with a DMA engine featuring parallel read and write channels, and an associated Runtime Bandwidth Regulators (RBR). The RBR is a device capable of throttling the bandwidth request of each DMA based on a throttling factor (*TF*) [18]. In addition, the platform is equipped with AXI performance monitors (PM) at each of the main memory controller input points [69, 70].

5.1.2 The throttling factor

Each of the RBRs regulates the bandwidth of its attached DMA engine as follows: the RBR throttles the engine's memory requests in such a way that the bandwidth demand of the engine drops to a given percentage of the original demand. One can set the desired percentage by setting appropriate values in some control registers of the RBR. *CBM* uses this mechanism to throttle accelerators. We call throttling factor (*TF*) the previous percentage. *CBM* regulates bandwidth by computing this throttling factor, as an integer number in $[0, 100]$ (extremes included), and by setting it on each RBR. This is achieved by using the function `set_TF_on_RBR(TF)`, which sets the value of the desiderd *TF* in a memory address constantly polled by the RBRs.

In the setup envisioned in this work, no individual bandwidth control is needed on accelerators. As a consequence, the bandwidth management policy presented can, and does, exercise any possible level of interference on the critical task, through a single, global *TF*, assigned to each DMA engine.

Hereafter we refer to this *TF* as just *the TF*.

5.1.3 System model

The system is comprised of the following actors, sharing a common main memory: a multicore CPU and a set of accelerators. One of cores of the CPU runs a *critical task* to completion. No other task runs on any other core. The only memory accesses we consider are those to main memory, for both the critical task, or equivalently the CPU, and the accelerators. For brevity, in the rest of the section we call just memory accesses, and bandwidth demanded/consumed by any of the actors, the accesses to main memory and the bandwidth of the main memory demanded/consumed by the actor. In the same vein, we call just memory interference the interference between these accesses.

Under the definition of task memory intensity I we previously provided (see Section 2.5.1), the only case where the execution of the critical task can be slowed down is if $I > 0$ and accelerators interfere with the task's memory accesses. In this respect, we define as *slowdown for an execution of the critical task*, the ratio between the duration of that execution and the duration that that execution would have had in isolation (i.e., in the absence of any memory interference). We assume that the critical task is associated with a *maximum tolerated slowdown* (*MTS*), equal to the maximum slowdown with which any of its executions must be completed for the QoS requirement of the task to be met.

We make the following general assumption: the execution time of the task in isolation is not known in advance, and may vary across executions. The only piece of knowledge available is then that the execution of the task may be slowed down only if the task demands some memory bandwidth during its execution (i.e., if $I > 0$), and if interference causes the task to receive less bandwidth than it demands. In this respect, the only part of the task that is slowed down by this reduction of bandwidth are its memory accesses, so the impact of memory bandwidth on the total slowdown of the task depends on what portion of the task is made of memory accesses. Yet the

memory intensity I of the task is unknown as well. Fortunately, the following weak relation still holds, between the bandwidth offered to the task and the maximum slowdown that the task may experience. This relation informs the main steps of *CBM*.

Theorem 1. *Regardless of the memory intensity I of the critical task, the latter is completed within its* MTS *if it enjoys, during its execution, an average memory bandwidth equal to at least a fraction $\frac{1}{\text{MTS}}$ of the average bandwidth it demands in isolation.*

Proof. Let T_{isol} and B_{isol} be, respectively, the execution time and the bandwidth demand of the task in isolation, and let T_I and T_{1-I} be the total execution time of the memory-access instructions of the task and of the rest of the task. If the task is executed in isolation, we have $T_I = T_{isol} \cdot I$ and $T_{1-I} = T_{isol} \cdot (1 - I)$.

Now suppose that, during an execution of the task, interference causes the average bandwidth of the task to become equal to $\frac{B_{isol}}{s}$, with $1 \leq s \leq MTS$. This reduction of the bandwidth affects the execution time of the fraction I of the task by a factor s, so $T_I = T_{isol} \cdot I \cdot s$. T_{1-I} remains unchanged instead. So the total execution time of the task becomes equal to $T_{isol} \cdot I \cdot s + T_{isol} \cdot (1 - I) = T_{isol} \cdot (I \cdot s + (1 - I))$. Then the total slowdown of the task is equal to $I \cdot s + (1 - I)$. Because $1 \leq s$ and $0 \leq I \leq 1$, the contribution of the second part of the equation is always going to be $\leq s$. Assuming maximum intensity ($I = 1$), this equates at most to a slowdown of $1 \cdot s + (1 - 1) = s \leq MTS$. □

On the accelerator side, we assume that the accelerator complex generates the traffic mixes that cause the highest possible interference on the critical task. On our test platform, this occurs in the following conditions: 1) the accelerator traffic is generated by any subset of the three accelerators that do not share any memory-controller port, as this set of accelerators evidently accesses memory with maximum parallelism, 2) both the DMA engines of each accelerator are active, with the first/second DMA engine generating an uninterrupted sequence of main-memory reads/writes. We call *1AccRW*,

2AccRW and *3AccRW* the total traffic resulting from having one, two or all three accelerators active (among the ones in the previous set). This traffic mimics a real, worst-case scenario in which accelerators happen to move data to and from their local memory during the lifespan of the critical task.

Finally, we assume that accelerators do not have any individual bandwidth requirement. The only mandatory requirement remains to guarantee the *MTS* of the critical task. Bandwidth management can achieve this goal by controlling (only) the cumulative memory bandwidth consumed by the whole accelerator complex.

5.2 The Bandwidth Manager (*CBM*)

The bandwidth manager we envision should be capable of reacting to very dynamic scenarios, where tasks with different QoS requirements come and go, and where the memory intensity I might change over different phases of the same task. The control mechanism should thus be adaptive, and leverage a tight control loop with a *monitor–decide–actuate* latency in the order of a few hundreds microseconds [71], to effectively react to the timing of a modern HeSoC task schedule.

Consider a time interval during which accelerators are generating memory traffic at full throttle (*3AccRW*). Assume that a critical task starts on the CPU: the task tolerates a given *MTS* and has a constant bandwidth demand equal to B_{isol} in isolation. The *MTS* requirement translates into a minimum bandwidth requirement of $\frac{B_{isol}}{MTS}$, by Theorem 1. Rather than simply adjusting the *TF* of the accelerators with small or proportional increments/decrements – in accordance with the CPU task use of memory – the ideal action would be to immediately select the *TF* that guarantees a target bandwidth equal to exactly the minimum bandwidth requirement indicated above. By doing so, accelerators would be throttled as little as needed, and the total bandwidth would be utilized as much as possible.

In this respect, if *CBM* had a mapping between any value of the target bandwidth and the corresponding *TF*, it could actually compute this optimal *TF* in one step. The intuition is illustrated in Fig. 5.2, for two basic

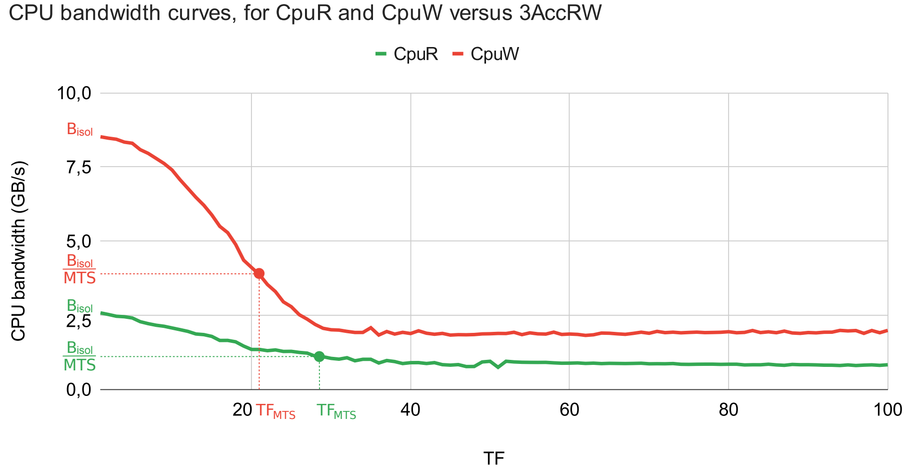


Figure 5.2: Key idea: a mapping between target bandwidth and *TF* allows to find the latter in one step.

traffic types: *reads* and *writes*, denoted as *CpuR* and *CpuW*. The bandwidth demand B_{isol} in isolation, for each traffic type, is equal to the leftmost value of the corresponding curve (the one for TF=0). For each curve and corresponding value of $\frac{B_{isol}}{MTS}$, the projection on the x-axis of the intersection between the curve and its horizontal line gives the value of TF_{MTS} to reach that target bandwidth. As pointed out by the figure, different traffic types lead to different TF_{MTS} for the same MTS.

Based on this intuition, our proposal relies on a set of bandwidth curves for a target HeSoC platform, retrieved via an ad-hoc *offline* characterization step for the most relevant combinations of CPU and accelerator traffic pat-

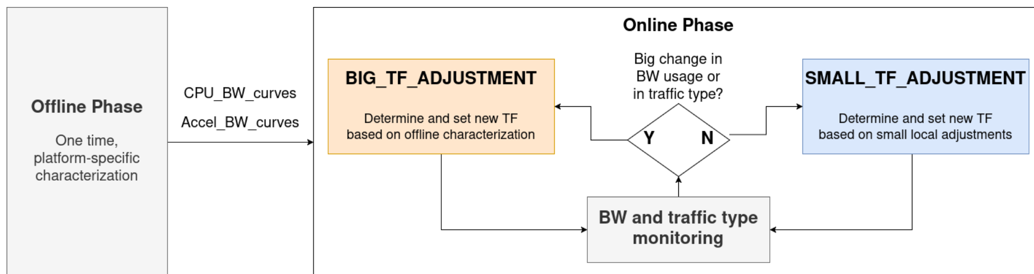


Figure 5.3: A scheme representing at a high level how *CBM* works.

terns. At runtime, an *online* policy monitors the bandwidth usage on the CPU and accelerator side and detects abrupt changes in the named bandwidth usage or in the traffic type. If no big change has occurred since the last iteration, the policy simply handles small *TF* increments or decrements. Otherwise, the policy determines which of the offline bandwidth curves best captures the current system state, rescales them as needed, and uses them to determine the *TF* that quickly adjusts to the abrupt change in memory usage. The block diagram of the proposed Characterization-based Bandwidth Manager (*CBM*) is shown in Fig. 5.3.

5.2.1 Offline phase

A key component of the proposed technique is the offline platform characterization step. The outcome of this stage is the set of bandwidth curves described in the previous section, which should capture system behavior for the most representative CPU/accelerator traffic combinations, and which are implemented as *look-up tables* (LUT). In this respect, any memory traffic combination can be as a point in a multidimensional space, where each axis represents one of the independent characteristics of the traffic. Main characteristics of interest for a modern HeSoC are:

- type of memory operations involved: reads, writes or both;
- percentage of each type of operations in the traffic;
- total bandwidth demanded by the traffic;
- spatial access pattern, considering sequential or random addressing (memory requests over time rely on sequentially increasing or randomly ordered addresses), contiguous or strided (consecutive memory requests target adjacent or distant memory words);
- size of individual transfers measured, e.g., in number of bytes to read or write;
- overall footprint of the memory region targeted by the sequence of operations.

It is of course impractical, if not impossible, to compute and store as part of a LUT a curve for each possible combination of values of the parameters listed above. For this reason, we only consider combinations that lead to traffics with the highest sensitivity to interference. These are the only traffics for which the need for regulation, and the beneficial effects of its application, are significant. We thus focus on sequential contiguous accesses, which have been shown to be the ones most affected by interference [30, 14, 22, 16, 15, 4]. On the accelerator side, we model varying memory activity (i.e., varying interference) only by considering a varying number of operational accelerators (one, two or three accelerators maximum, matching the characteristics of the target hardware). The traffic these accelerators individually generate can be read-only, write-only or read-write, but always of the sequential and contiguous type, matching the typical DMA-based memory operation of such accelerators.

5.2.1.1 Traffic generators

To conduct the offline characterization under all the relevant CPU traffic types, we use *CpuR*, *CpuRW*, and *CpuW* on the CPU (introduced in Section 2.5.1). Note that for *CpuRW* the traffic leverages two parallel flows of reads and of writes, each targeting a different memory region.

Varying memory intensity (Sec. 5.1.3) in the CPU traffic generator can be modeled by using a changing the *cops* parameter of the benchmarks.

To model the relevant accelerator traffic types, instead, we design a similar synthetic benchmark. The same three basic contiguous types are considered: (i) read-only (*AccR*); (ii) write-only (*AccW*); (iii) read/write (*AccRW*). In particular, accelerator traffic is always contiguous, which more faithfully captures the burst transfers typically employed by DMA engines. Here, each DMA engine is controlled by a local soft-core, which executes an infinite loop that continuously enqueues 512KB-sized burst operations of the selected type between main memory and local Block RAM. Memory intensity on the accelerator side is the number of accelerators/DMAs involved: one (*1Acc*), two (*2Acc*), or three (*3Acc*). Note that in the following we often fuse accelera-

tor traffic type and intensity in the same notation (e.g., three accelerators generating read/write requests are indicated as *3AccRW* traffic).

5.2.1.2 Data structure and characterization algorithm

The look-up operation for the expected bandwidth usage on CPU core and accelerators leverages two five-dimensional matrices, `CPU_BW` and `accel_BW`. The five dimensions are indexed by, respectively: (i) accelerator traffic type (among *AccR*, *AccW* and *AccRW*); (ii) accelerator intensity (among *1Acc*, *2Acc* and *3Acc*); (iii) CPU traffic type (among *CpuR*, *CpuW* and *CpuRW*); (iv) CPU intensity (among discrete increments of 0.2 in the range $[0, 1]$), and (v) *TF* (among unit increments in the range $[0, 100]$). It is noteworthy that specifying each of these five parameters provides a unique bandwidth value. However, a row in each of the matrices (when the *TF* value is unspecified) contains an entire bandwidth curve, as explained in Section 5.2. For example, Fig. 5.2 shows the curves contained in the two rows of `CPU_BW` that are indexed by accelerator traffic type and intensity *3AccRW*, CPU traffic type *CpuR*/*CpuW*, and CPU intensity 1.

Both matrices are populated by Algorithm 5. The generators on the accelerators are started and stopped right before and right after the generator on the CPU is started and stopped, respectively. While accelerators are interfering, each of the three CPU traffics is generated at one of five possible values of intensity, and for one of the possible values of *TF*. Both CPU and accelerator bandwidths are measured over the last bandwidth-measurement period, and stored in `CPU_BW` and `accel_BW`, respectively. In this respect, the first point of each row in `CPU_BW` (for *TF*=0) represents the bandwidth demanded by the critical task in isolation.

5.2.1.3 Shape and properties of CPU and accelerator bandwidth curves

We discuss here relevant pieces of information extracted from the analysis of the offline bandwidth curves that are provided as an input to the online phase.

Algorithm 5: Characterization algorithm.

```
1  Function set_TF(TF)
2      foreach accel in set_of_active_accelerators
3          set_TF_on_RBR(accel, TF)

4  Function do_BW_characterization()
5      foreach AT in {AccR, AccW, AccRW}
         //sweep accelerator traffic type
6          foreach AI in {1Acc, 2Acc, 3Acc}
             //sweep accelerator intensity
7              start_accel_generators(AT, AI)
8              foreach CT in {CpuR, CpuW, CpuRW}
                 //sweep CPU traffic type
9                  foreach CI in {.2, .4, .6, .8, 1}
                     //sweep CPU intensity
10                     foreach TF in {0, 1, 2, ..., 100}
                         //sweep throttling factor
11                         set_TF(TF)
12                         start_CPU_generator(CT, CI)
13                         sleep(BW_measurement_period)
14                         CPU_BW[AT, AI, CT, CI, TF] =
                             measure_CPU_BW()
15                         accel_BW[AT, AI, CT, CI, TF] =
                             measure_total_accel_BW()
16                         stop_CPU_generator()
17             stop_accel_generators()
```

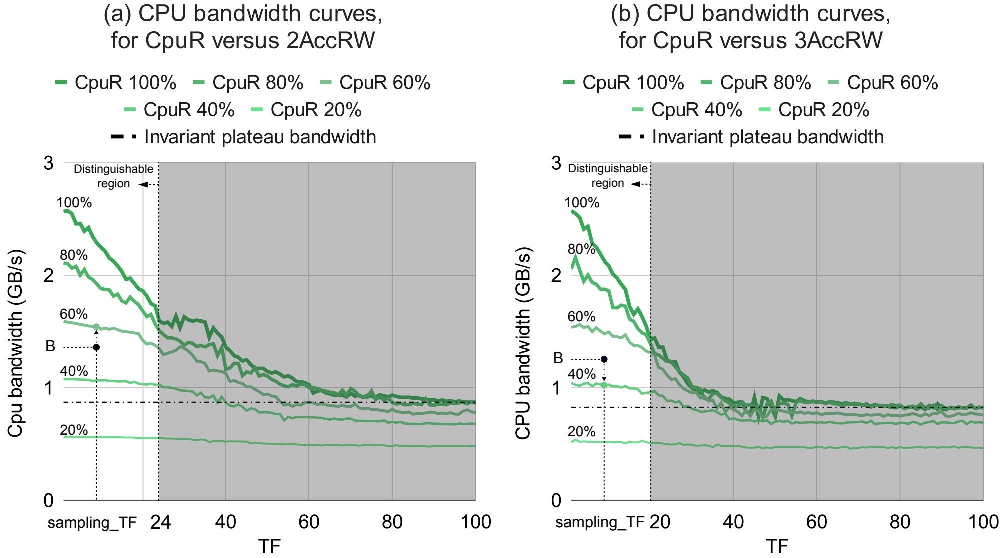


Figure 5.4: Bandwidth curves for *CpuR* against *2AccRW* and *3Acc*. Each figure also shows the closest curve found for a given value of `critTask_BW`, with the latter denoted as just B, to reduce clutter.

When it comes to the CPU curves, Figures 5.4a and 5.4b show the characterization for *CpuR* versus *2AccRW* and *3AccRW*, respectively (note that intensities are expressed as percentages, instead of numbers in $[0, 1]$).

First, we observe that highest-intensity curves are *distinguishable* from each other only in the left region of the graph. Accordingly, we define *CPU_max_dist_TF* for a given type of CPU traffic as the highest value of *TF* such that, for any pair of curves, the difference between the values of the curves for that *TF* is higher than 1% of the difference between the highest and the lowest bandwidth values measured over all the curves. The threshold is set to 1%, to be robust against unavoidable noise in bandwidth measurements. This information is needed to find the curve in `CPU_BW` that most closely approximates the actual *bandwidth curve* of the critical task. Note that such value is easily identified as part of the offline characterization. In particular, Fig. 5.4b illustrates the traffic combinations for which, on the target platform, we observe the smallest *CPU_max_dist_TF* (*TF*=20). This *TF* value thus identifies a valid limit for the distinguishable CPU region

for every traffic combination.

Second, some curves are not distinguishable in the right region of the graph, and there they exhibit a plateau where they are virtually independent of *TF* (Figures 5.4a and 5.4b). The cause of this invariant plateau is system bandwidth saturation, and only affects traffic with sufficiently high memory intensity. For lower intensities, the bandwidth curves exhibit a different saturation behavior: the plateau is still present, but the associated bandwidth value decreases with the memory intensity. For very low memory intensity there is no bandwidth saturation, as it is intuitive, and bandwidth curves approximate straight lines. As an outcome of the characterization, we also flag each curve as having/not having an invariant plateau through a boolean parameter `has_invariant_plateau(curve)`.

The value of the *TF* at which the plateau begins is also relevant to the online phase of *CBM*. This value is computed as the smallest *TF* such that, from that point on, the value of the curve does not change by more than 3% with respect to the value of the curve at that point. The 3% threshold complies with the fluctuations that, in our experiments, curves happened to exhibit in their plateaus when it comes to consecutive measurements. Each curve has this information stored in an integer parameter `plateau_start_TF(curve)`.

Focusing on accelerator curves, Fig. 5.5 shows the outcome of the offline characterization for *1AccRW* (thin lines), *2AccRW* (medium-thickness lines) and *3AccRW* (thick lines) competing against CPU traffic of type *CpuR* (yellow curves), *CpuW* (purple curves) and *CpuRW* (blue curves). All CPU traffics are at 100% intensity. Identical plots were also produced for accelerator traffic types *AccR* and *AccW*, considering all intensities (*1Acc*, *2Acc* and *3Acc*). These plots are not shown because they don't show different effects than those illustrated here.

The notion of a *distinguishable* region where curves belonging to different accelerator intensities are clearly separate still holds. In addition, another important property must be introduced: *independence* from the CPU traffic type. Through all the leftmost part of the plot only three curves are visible out of the nine drawn. Accelerator intensity is the parameter that deter-

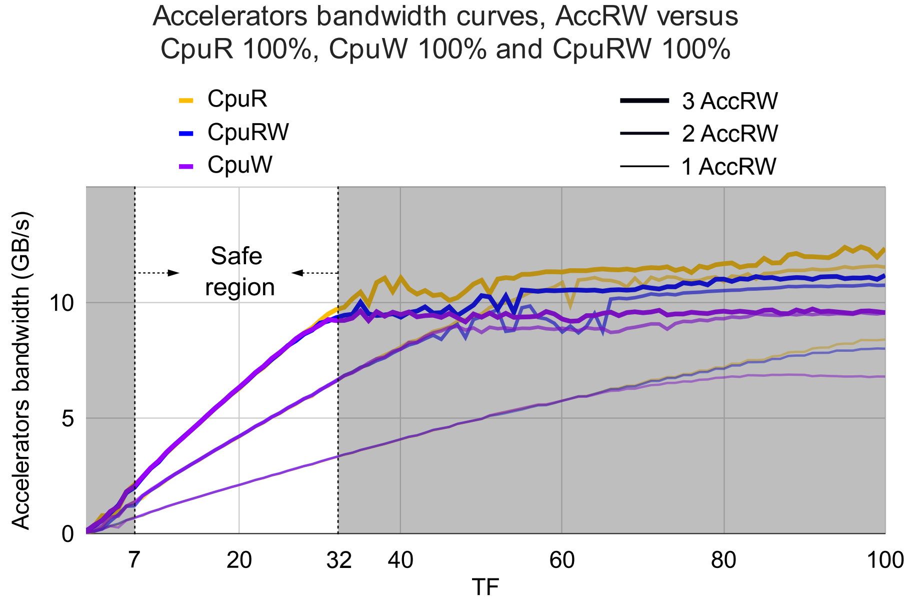


Figure 5.5: Accelerators bandwidth curves with different Cpu tasks running in the background.

mines significant variations in the bandwidth consumed by the accelerators, independent of the interfering CPU traffic type. In other words, CPU traffic type affects bandwidth usage by a negligible amount in this region, to the point where curves are clustered three by three, virtually collapsing in a single curve per accelerator intensity value.

Picking a *TF* value that belongs to the region where accelerator curves are both distinguishable and independent from CPU traffic type has the important property of enabling the one-step *TF* selection idea described at the beginning of Sec. 5.2. We thus call this *TF* interval a *safe region* for *TF* measurement, which is delimited by both a left (*accel_min_safe_TF*) and a right limit (*accel_max_safe_TF*). Out of these bounds the curves enter the saturation region and become indistinguishable and/or dependent from CPU traffic type.

Again, both limits are easily identified as part of the offline characterization. In particular, Fig. 5.5 identifies the traffic combinations for which, on the target platform, we observe the largest *accel_min_safe_TF* (*TF*=7) and

the smallest *accel_max_safe_TF* (*TF*=32). Using these *TF* values as limits to a *safe* accelerator region thus holds valid for every traffic combination.

As a final, relevant note we report that by observing the sum of CPU and accelerator bandwidths for every traffic combination an important property is highlighted: the total bandwidth always grows with *TF*. That is, for any value of *TF*, accelerator bandwidth increases with *TF* more than the CPU bandwidth decreases.

5.2.2 Online Phase

Algorithm 6 illustrates the functioning of the online phase of *CBM*. To simplify pseudocode, we assume that all variables referenced in any of the algorithms reported in this thesis are global. Occasionally, some variables are however passed as arguments to some function, in order to highlight the main

Algorithm 6: *CBM* general algorithm.

```
1  old_critTask_BW ← 0
2  cur_TF = 100  // start with full throttle for accelerators
3  Function update_throttling_factor()
4      critTask_BW = measure_CPU_BW()
5      BW_Delta = abs(critTask_BW - old_critTask_BW)
6      if (BW_Delta >= large_Delta or traffic_mix_has_changed()) {
7          compute_target_BW_correction()
8          do_big_TF_adjustment()
9      } else { // small BW change and no traffic-mix change
10         compute_target_BW_correction()
11         do_small_TF_adjustment()
12     }
13     set_TF(cur_TF)
14     old_critTask_BW = critTask_BW
15 Function CBM_main()
16     while(true) {
17         update_throttling_factor()
18         sleep(BW_measurement_period)
19     }
```

inputs that the function works on. The online phase of *CBM* continuously monitors the bandwidth used by the critical CPU task (`critTask_BW`), and ensures that the quality-of-service requirement expressed via its MTS is respected by throttling the cumulative bandwidth demanded by all accelerators (the *throttling factor TF*, described in Sec 5.1.2).

CBM continuously invokes the function `update_throttling_factor()` then sleeps for a pre-defined time period (`BW_measurement_period`) in an infinite loop. The whole decision process inside `update_throttling_factor()` starts by measuring `critTask_BW`. For this measurement to be reliable, it has to be taken over a long-enough time interval (for the hardware registers to properly update), hence the *sleep*. In particular, bandwidth measurements fluctuate more and more as the total bandwidth approaches saturation. To get a good tradeoff between latency and precision, we set `BW_measurement_period` to the minimum period for which, on the platform at hand, any sequence of repeated measurements, in the same conditions, has a standard deviation of at most 2%. This hold true even at the highest level of saturation of the memory bandwidth.

Note that this main loop dictates the overall speed of the technique. The total time period with which *CBM* updates the throttling factor is equal to the sum of the bandwidth-measurement period and the execution time of `update_throttling_factor()` (see Sec. 5.3).

After measuring `critTask_BW`, `update_throttling_factor()` checks whether it, or the traffic mix, has changed significantly since the last iteration. Traffic mix changes include switches in both the CPU and the accelerator traffic types, among *read*, *write* and *read/write*. If either of the significant changes has happened, then both the bandwidth demand of the critical task and the interference from accelerators may have changed. In this case the function `do_big_TF_adjustment()` is invoked, where the bandwidth curves obtained in the offline stage are inspected to quickly determine possibly big changes in *TF*, according to the one-step change idea shown in Fig. 5.2. If, on the contrary, the bandwidth change has been small and the traffic mix has not changed since the last iteration, the current CPU bandwidth is still close to convergence to the target bandwidth, and only a small change in the

throttling factor may be needed. This is performed by *do_small_TF_adjustment()*, which implements small incremental adjustment in a linear feedback control fashion. A detailed description of these procedures is provided in Sec. 5.2.2.1.

Note that in both cases a correction factor is applied to the target BW via a call to the *compute_target_BW_correction()* function, before updating the throttling factor. This correction acts as a sort of bandwidth reclaiming mechanism, and is foreseen so as to balance possible bandwidth losses or extra gains that may have occurred in the previous update periods. Details are provided in Sec. 5.2.2.2.

5.2.2.1 Determining the optimal ***Throttling Factor***

Algorithm 7 shows the procedures to update the *TF* in case of big or small adjustments needed. Starting from the latter, `do_small_TF_adjustment()` simply implements the increment/decrement of the *TF* for the linear feedback-control case, using the *corrected* target bandwidth as a reference.

Algorithm 7: *TF* update functions.

```
1  target_BW = 0
     // left to zero until the BW of the critical task is first measured
2  target_BW_correction = 0
3  Function do_big_TF_adjustment()
4  |   closest_BW_curve = find_closest_curve()
5  |   scaling_factor = critTask_BW / closest_BW_curve[safe_TF])
6  |   target_BW_curve = rescale_curve(closest_BW_curve,
   |     scaling_factor)
7  |   // determine new TF
8  |   cur_TF = get_highest_TF(target_BW +
   |     target_BW_correction, target_BW_curve)
9  Function do_small_TF_adjustment()
10 |   if (critTask_BW > target_BW + target_BW_correction)
11 |       cur_TF++
12 |   else
13 |       cur_TF - -
```

The function `do_big_TF_adjustment()` first invokes `find_closest_curve()` to retrieve the offline bandwidth curves that best approximate the CPU and accelerator behavior for the current system state. Then proper rescaling of the curves is applied to refine the offline model, so as to more closely match the measured critical task bandwidth. This step is performed by the `rescale_curve()` function. Finally, the optimal *TF* is extracted by these curves via the `get_highest_TF()` function.

`find_closest_curve()` works as shown in Algorithm 8. The key idea of the one-step *TF* selection is that for every target HeSoC it is possible to statically determine, at the end of the offline characterization stage, a range of *TF* values for which both the CPU and the accelerator bandwidth curves are all clearly distinguishable and accelerator intensity is independent from CPU traffic type. This *global safe region* can be obtained as the intersection of the CPU distinguishable region and the accelerator safe region discussed in Sec. 5.2.1.3. It is then sufficient to select a "safe" *sampling_TF* value within this region to measure: (i) the CPU (i.e., the critical task) bandwidth and traffic type; (ii) the accelerator(s) bandwidth and traffic type. In this operating condition it is easy to look-up for the target curve searching among only the LUT entries matching the current combination of accelerator and CPU traffics. For our target HeSoC, as discussed in Sec. 5.2.1.3, the global distinguishable region is found between the lower bound $lower_safe_TF = accel_min_safe_TF = 7$ and the smaller of the upper bounds $upper_safe_TF = CPU_max_dist_TF = 20$. Any *TF* value within this region constitutes a "safe" *sampling_TF*. To simplify the algorithm logic we define it as a single value *sampling_TF*=10, near the middle of the global distinguishable region. Note that, as an additional optimization, the algorithm checks, at line 12, if the previous *critTask_BW* was already measured within the safe region. In this case it is not needed to repeat the measure.

Once the "safe" *critTask_BW* has been measured, we contextually determine CPU and accelerator traffic type, as well as accelerator intensity. `get_CPU_traffic()` and `get_accel_traffic()` determine the CPU and ac-

Algorithm 8: Find closest curve.

```
1  Function get_CPU_traffic()
2      // Returns CPU traffic type based on CPU load/store
3      // count retrieved from the system PMU.
4  Function get_accel_traffic()
5      // Returns accelerator traffic type based on DMA
6      // load/store count retrieved from the system PMU.
7  Function get_accel_intensity(accel_BW)
8      // Returns accelerator intensity based on which of the accelerator
9      // BW curves most closely matches accel_BW at sampling_TF.
10 Function find_closest_curve()
11     // if last BW measure was taken out of safe region,
12     // measure BW again at safe TF
13     if ((cur_TF < lower_safe_TF) or (cur_TF > upper_safe_TF){
14         cur_TF = sampling_TF
15         set_TF(cur_TF)
16         sleep(BW_measurement_period)
17         critTask_BW = measure_CPU_BW()
18     }
19     //determine CPU and accelerator traffic type
20     CT = get_CPU_traffic()
21     AT = get_accel_traffic()
22     AI = get_accel_intensity(measure_accel_BW())
23     foreach intensity in {.2, .4, .6, .8, 1}
24         curr_diff = abs(CPU_BW[AT, AI, CT, intensity,
            sampling_TF] - critTask_BW)
25         if (curr_diff < min_diff)
26             closest_intensity = intensity
27             min_diff = curr_diff
28     return CPU_BW[AT, AI, CT, closest_intensity]
```

celerator traffic type by measuring, and comparing, the numbers of reads and writes, through hardware performance counters [13, 24, 29]. The function `get_accel_intensity()` finds the curve that, at `sampling_TF`, yields the closest value to the bandwidth generated by the accelerators over the last bandwidth-measurement period, then returns the associated intensity.

Finally, only the CPU curves for the given CPU/accelerator traffic mix, and for the specific accelerator intensity and *sampling_TF* are selected. By sweeping all CPU intensities the curve which more closely matches *critTask_BW* is finally extracted.

Figures 5.4a and 5.4b show an example outcome of this search for two cases. In the first case, the closest curve is the one above `critTask_BW`, while in the other case, the curve below. To reduce clutter, `critTask_BW` is denoted as just B in the figures. The examples show a common case in which the curve returned by `find_closest_curve()` doesn't perfectly match the actual bandwidth curve of the critical task, as the characterization only includes the most relevant traffic combinations. Yet, the parameters of the sampled curves have been chosen to get curves as representative as possible for any memory traffic (Sec. 5.2.1). Therefore, if a non-coincident curve is found in the search, that curve is still expected to have about the same shape as the actual bandwidth curve of the critical task. So proper rescaling is enough to get the latter from the former. This is the step performed by `rescale_curve()`, shown in Algorithm 9.

`rescale_curve()` considers two alternatives. First, the closest curve does not have an invariant plateau (Sec. 5.2.1.3, Fig. 5.4b). For this alternative, `rescale_curve()` assumes that the estimated curve does not either, and performs a simple rescaling. This yields a correct bandwidth curve, as shown in Fig. 5.6a for a sub-case where the closest curve must be scaled down. The estimated curve sits correctly between the closest curve and the lower-intensity curve in the characterization.

For the other alternative, a plain rescaling would produce an anomalous curve. The problem is shown in Fig. 5.6b, for a scaling-up subcase. A simply scaled-up curve would have a higher plateau than the maximum pos-

Algorithm 9: Simple rescale and rescale to join.

```
1  Function rescale_and_join(target_BW_curve,
     reference_BW_curve, scaling_factor)
2      plateau_start_TF = plateau_start_TF(reference_BW_curve)
3      plateau_BW = reference_BW_curve[plateau_start_TF]

4      for (tf = sampling_TF + 1 ; tf <= plateau_start_TF; tf++)
5          cur_diff = reference_BW_curve[tf] - plateau_BW
6          target_BW_curve[tf] = plateau_BW + scaling_factor *
             cur_diff

7      for (tf = plateau_start_TF + 1 ; tf <= 100; tf++)
8          target_BW_curve[tf] = reference_BW_curve[tf]

9  Function rescale_curve(reference_BW_curve, scaling_factor)
10     for (tf = 0 ; t <= sampling_TF; tf++)
11         target_BW_curve[tf] = scaling_factor *
             reference_BW_curve[tf]

12     if(not has_invariant_plateau(reference_BW_curve))
13         // just finish scaling the curve up or down
14         for (tf = sampling_TF + 1 ; tf <= 100; tf++)
15             target_BW_curve[tf] = scaling_factor *
                 reference_BW_curve[tf]
16     else
         // reshape the curve to get a rescaled version with the same plateau
17         rescale_and_join(target_BW_curve, reference_BW_curve,
             scaling_factor)
```

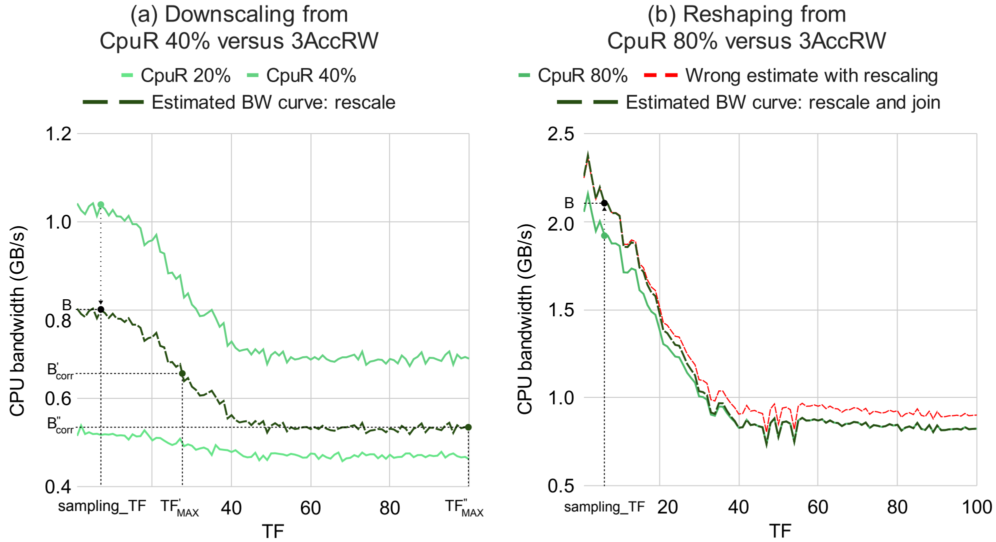


Figure 5.6: Examples of: (a) downscaling from CpuR 40% and (b) upscaling and joining from CpuR 80%. Subfigure (a) also shows two examples of computation of the throttling factor, as a function of two possible values, $B^{'}_{corr}$ and $B^{'}_{corr}$, of the corrected target bandwidth.

sible (Fig. 5.4b). Specularly, for the scaling-down subcase, the scaled-down curve would have a lower plateau than the closest curve. This would would be wrong as well, because the estimated curve has most likely an invariant plateau (equal to that of the closest curve).

To get a correct curve, the scaling factor must decrease more and more, after `sample_TF`, as the curve approaches the beginning of the plateau. Until it becomes equal to one (i.e., no scaling) at the beginning of the plateau. Then it must remain equal to one, until the end of the curve. This is performed by the function `rescale_and_join()` in Algorithm 9. The result is shown in Fig. 5.6b.

Once the rescaled curve is computed, the optimal TF can be determined by looking at the point where the target bandwidth value intersects the curve, as shown in Fig. 5.3. However, when the system is operating in saturation (the rightmost region of the plots), increasing the bandwidth available to the accelerator only slightly increases overall bandwidth us-

age, without further slowing down the CPU critical task. For this reason, `do_big_TF_adjustment()` returns the highest *TF* for which the estimated bandwidth curve yields the target bandwidth (`get_highest_TF()`). This *TF* is found by, first, getting the *TF* at the intersection between the target bandwidth and the curve, and, then, shifting the value as far as possible to the right, until the bandwidth stays within 1% of the initial value. Fig. 5.6a shows this last step, for two extreme cases. The target bandwidth is denoted as $B^{'}_{corr}$ in the first case, and as $B^{''}_{corr}$ in the second case. In the first case, `do_big_TF_adjustment()` returns $T^{'}_{MAX}$, which is just the projection on the x-axis of the intersection between $B^{'}_{corr}$ and the estimated bandwidth curve. In the second case, $B^{''}_{corr}$ is so low that it falls on the plateau of the estimated curve, and `do_big_TF_adjustment()` can afford to set $T^{''}_{MAX}$ even to 100. This yields the maximum possible gain in total bandwidth utilization.

5.2.2.2 Correcting the target bandwidth

There are circumstances in which the CPU critical task might consume less bandwidth than the target bandwidth computed by *CBM*. Fig. 5.7a shows the consequences of such an instance. Suppose that, at time t_0, the critical task receives less than the target bandwidth computed by `update_target_BW()` (in the figures we amplify the error, to show the problem more clearly). During $[t_0, t_1]$, *CBM* corrects the error on the throttling factor until the critical task receives the target bandwidth. Yet the average bandwidth has lowered in the meantime. So, the average bandwidth of the task happens to be lower than the average target bandwidth by time t_1. If the task terminated at time t_1, Theorem 1 would not be met, and the *MTS* would not be guaranteed.

Let's assume that the task continues, but at time t_1, as with some of the real applications in our experiments, the bandwidth demand of the task is so low that the latter suffers from negligible interference in $[t_1, t_2]$. Therefore, for any possible value of the throttling factor, the task gets all the (low) bandwidth it demands. This bandwidth is of course higher than the target bandwidth computed by *CBM*. So the average bandwidth of the task decreases more slowly than the average target bandwidth, until it becomes

equal to the latter. Then it becomes higher and higher than the target bandwidth.

Finally, in $[t_2, t_3]$, the task asks again for more bandwidth, and *CBM* raises the target bandwidth consequently. In $[t_2, t_3]$, *CBM* succeeds in providing the task with the target bandwidth. However, since the task has received more than its due average bandwidth by time t_2, it keeps receiving more than its target average bandwidth in $[t_2, t_3]$ as well. The problem now is the opposite of that at time t_1: in $[t_2, t_3]$ *CBM* has provided the task with more than its due bandwidth, according to Theorem 1. *CBM* could have set a higher throttling factor, without breaking time guarantees for the task. This would have boosted the total utilization of the memory bandwidth, because, as noted at the end of Sec. 5.2.1.3, the total utilization of the memory bandwidth always grows with the throttling factor.

To avoid this type of deviations, we enrich *CBM* with a mechanism to correct the target bandwidth when computing the throttling factor. This corrected bandwidth is computed so as to constantly tend to provide the critical task with an average memory bandwidth $B_{MTS} = \frac{B_{isol}}{MTS}$, where B_{isol} is the average bandwidth demanded by the critical task (over its lifespan) in isolation. This in turn guarantees that the task meets its *MTS* by Theorem 1.

The function `compute_target_BW_correction()` is reported in Algorithm 10. It computes a corrected bandwidth with the following property: if the current average bandwidth of the critical task is lower/higher than the average target bandwidth (with both averages computed from the beginning of the memory activity of the task), then the average bandwidth of the task becomes equal to the average target bandwidth if the task gets a bandwidth equal to the corrected bandwidth in the next update period. The correction factor is then obtained from this corrected bandwidth.

Fig. 5.7b shows the effects of the correction, assuming that the critical task has the same bandwidth demand, over time, as in Fig. 5.7a. The curve for the target bandwidth is consequently the same as in Fig. 5.7a, so we do not report it in Fig. 5.7b as well, to reduce clutter.

The use of a corrected target bandwidth lets the task recover, during the

Algorithm 10: Target BW correction functions.

```
1  avg_BW_sum = 0
     // sum of measurements of critical task BW, used to compute avg
2  avg_target_BW_sum = 0
     // sum of samples of target BW, used to compute avg
3  num_BW_measures = 0
     // number of measurements of critical task BW
4  Function compute_target_BW_correction()
5      avg_BW_sum = avg_BW_sum + critTask_BW
         // add last BW measurement
6      avg_target_BW_sum = avg_target_BW_sum + target_BW
         // add last BW measurement
7      num_BW_measures++
         // increment number of BW measurements
8      avg_target_BW = avg_target_BW_sum /
         num_BW_measures
9      // corrected_target_BW is the BW value such that,
10     // if the critical task enjoys this bandwidth in the next period,
11     // then the average bandwidth of the task becomes
12     // equal to the average target_BW a the end of that period
13     corrected_target_BW = (num_BW_measures + 1) *
         avg_target_BW - avg_BW_sum
14     target_BW_correction = corrected_target_BW - target_BW
```

second half of $[t_0, t_1]$, the loss of bandwidth that it suffers during the first half of the time interval. So, if the task terminated at time t_1, the average bandwidth that it enjoyed up to time t_1 would guarantee that the task met its *MTS*. On the opposite end, during $[t_2, t_3]$, *CBM* provides the task with less than the target bandwidth, until the average bandwidth of the task becomes equal to the average target bandwidth.

5.3 Experimental Evaluation

In this section, we analyze the cost and the performance of *CBM* in terms of both guaranteeing the *MTS*, and providing good system bandwidth uti-

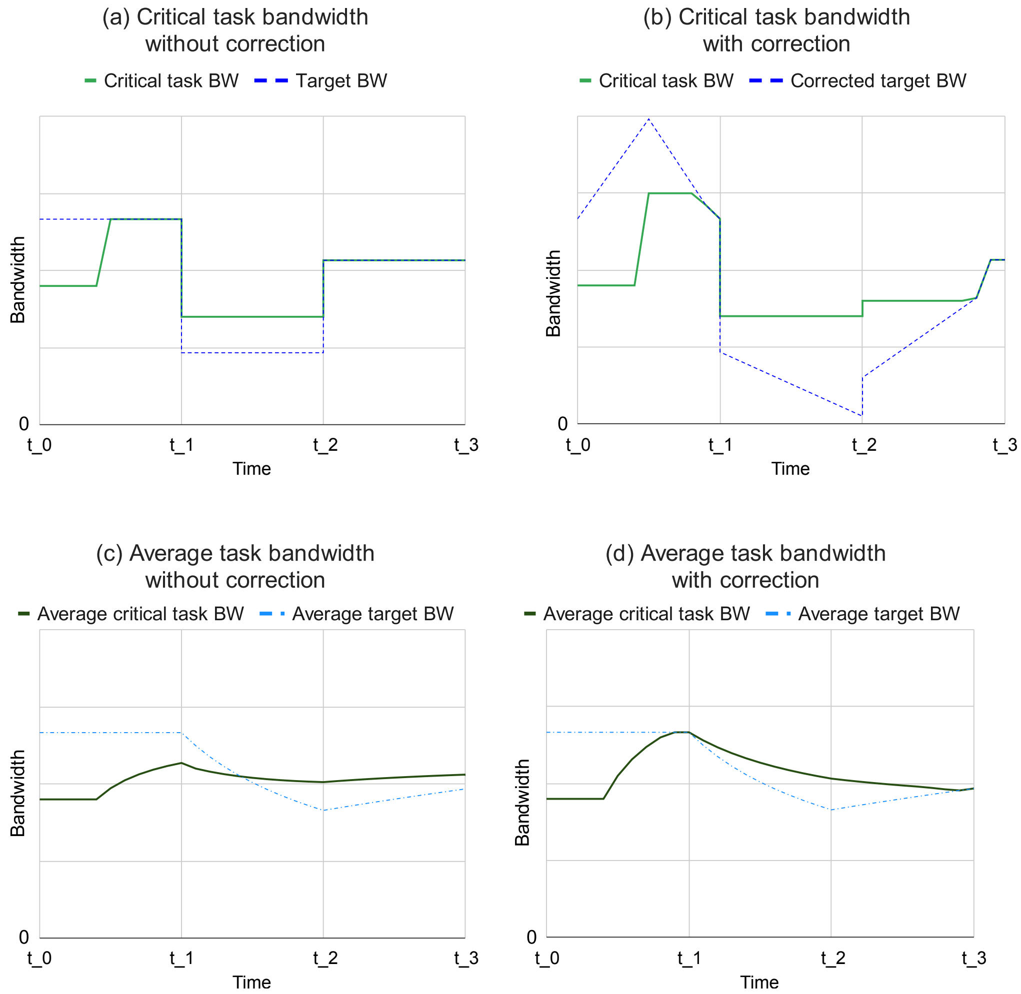


Figure 5.7: Figure (a) shows the consequences of performing no correction on the target bandwidth (failure to provide the critical task with no less and no more than the average target bandwidth), while figure (b) shows how the correction addresses this issue. Figure (c) and figure (d) show how the average critical task bandwidth compares to the average target bandwidth, as time passes.

lization. First, we provide numbers that quantify the cost (overhead) of *CBM* and, consequently, its operating speed. Second, we analyze how *CBM* performs when real benchmarks (the Polybench suite, introduced in Section 2.5.2) are used as *critical tasks*, for various *MTS* levels. Then, we study the ability of *CBM* to adapt to dynamic workloads by instantiating a benchmark made of frequent changes in traffic type, at the same *MTS* levels as before. In particular, we study how the frequency of traffic type changes affects the ability of *CBM* to guarantee the *MTS*. All evaluations we performed were done on the *Xilinx ZU9EG*. For each evaluation, we report the average value across 15 independent runs, as we observe very low variance across the measurements we obtained.

5.3.1 *CBM* implementation, overhead and time granularity

We implemented *CBM* on the *Xilinx ZU9EG* as a pair of components: A user-space program implements the offline characterization (Sec. 5.2.1). The online phase described in Algorithm 6 is implemented as a Linux kernel module, with the `sleep` function implemented by programming a high-resolution timer. The latter is pinned on a different core than that executing the critical task.

Being the online phase implemented by a kernel module, bandwidth values must be integer numbers. Assuming a 64-bit precision for memorizing bandwidths (as they can exceed 4 GB/s), the total memory occupied by the CPU and the accelerator curves is equal to 225 KBs.

Reported below the time needed for the various components of the online phase of the algorithm:

- Duration of the bandwidth-measurement period (to get reliable bandwidths and ratios of reads and writes): 300 μs
- Execution time of *update_throttling_factor()*: 5 μs
- Resulting duration of the update period: 305 μs

The online policy was set to perform its periodic checks every 300 μs due to the limitations in the frequency with which the bandwidth can be measured using the AXI Performance Monitors. In particular, we observe that measuring the bandwidth more often than every 250μs on the Xilinx ZU9EG leads to either skewed measurements (with the average bandwidth of a task increasing as the measurement frequency increases) or fluctuations in the values measured. The execution of the code of the policy is in the order of microseconds. However, the ability of *CBM* to react to traffic changes is dependent on how quickly it can measure the bandwidth of a task. On the *Xilinx ZU9EG*, the contribution of the measurement period is significantly higher than the contribution of the policy itself, leading to lowered reaction times. We installed on the test platform an Ubuntu 20.04.2, with a Linux 6.6.0 for Xilinx. The distribution has a minimal server configuration (no GUI or audio), to minimize OS noise. The critical task (defined in Sec. 5.1.3) is executed as a Linux process, with maximum priority and pinned on one of the cores, to prevent any de-scheduling or migration.

5.3.2 MTS guarantee - Polybench

Polybench benchmarks do generate a dynamic traffic, which is a valid testbed for *CBM*.

First, we chose to ascertain the performance of *CBM* while running 3AccRW on the accelerators. This was done to measure how well the control scheme is able to reach our goals of not exceeding the *MTS* while keeping hardware usage high even under the most demanding conditions. We chose to have the Polybench suite [50] execute as the *critical task*, since they represent generic workloads with really varied amounts of memory usage. We then set a wanted *MTS* for *CBM*, and measured the execution time of the Polybench with *CBM* active as the accelerators were creating interference. Finally, we compared the obtained results with the execution time of the Polybench while running without being subject to any interference, to measure how close the slowdown of the task that *CBM* provides is to the *MTS* we wanted.

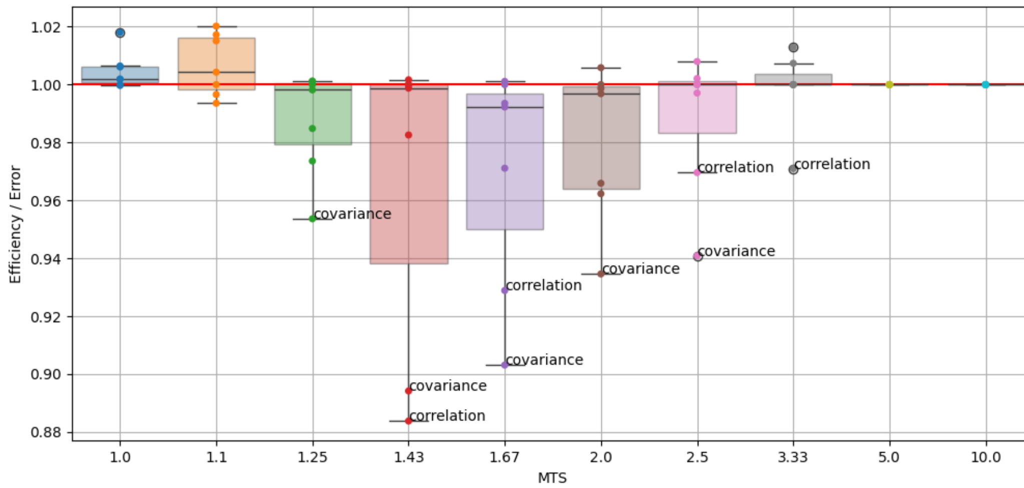


Figure 5.8: Boxplot representing how *CBM* performs when the Polybench benchmarks are executed as the *critical task*, with three accelerators creating interference. On the X axis, the requested *MTS*, obtained by converting decreasing BW% levels to *MTS*. On the Y axis, the ratio between the obtained slowdown and the *MTS*. In red, the ideal target ratio (1.00), which we call maximum *Efficiency*.

Fig.5.8 is a box plot which shows how *CBM* behaves for *MTS* values obtained by converting BW% requests decreasing by 10% each to slowdowns (e.g. BW 100% == 1.0×, BW 90% == 1.1×, BW 80% == 1.25×). The resulting *MTS* are reported on the X axis. On the Y axis (*Efficiency / Error*) is the ratio between the slowdown achieved by the control scheme and the target *MTS*. In cases where the program cannot be slowed down as much as the target *MTS*, the ratio between the slowdown achieved by the control scheme and the maximum slowdown that the program can be subject to is reported instead. In **Fig.5.8**, the resulting ratios for the Polybench are grouped in boxes depending on the *MTS* requested, with the greatest outliers being highlighted. In particular, a ratio > 1.00 means that the CPU task was slowed down too much, and the *MTS* was exceeded. This is called *Error* in the graph, and it is represented by the CPU tasks exceeding the target (red line). A ratio < 1.00 represents that the slowdown was lower than the *MTS*. This is called *Efficiency* in the graph. While a lower *Efficiency* is not as problematic as an *Error*, it is important to note that a ratio lower than

the 1.00 target means that the accelerators could have been throttled less, meaning there was system bandwidth underutilization.

With this in mind, there are two observation that can be done from the results reported in **Fig.5.8**. First, the *Error* is always less than 2%, even when the *MTS* is low (1.0× and 1.1×). This means that *CBM* is able to properly guarantee that a task is not slowed down more than the *MTS*. Second, there are certain tasks for which *CBM* throttles the accelerators too much. The two applications most subject to this phenomenon are *covariance* and *correlation* (with a minimum *Efficiency* of 0.89). When looking into them, we noticed that they tend to have small (but repeated) periods of time with low memory bandwidth request, followed by longer sections of high memory bandwidth request. *Covariance* and *correlation* are the benchmarks that present this pattern the most. This pattern triggers a pathological case in our online policy: the problem is that the low traffic sections are long enough to trigger a call to `do_big_TF_adjustment()`. By the time `do_big_TF_adjustment()` is done, the benchmark has switched back to a high traffic section, meaning that *CBM* has set the wrong *TF*, and needs to call `do_big_TF_adjustment()` yet again. This negatively impacts on the ability of *CBM* to match the requested *MTS*. *CBM* spends too much time with the *TF* set to *sampling_TF* (due to the multiple calls to `do_big_TF_adjustment()`).

Having the *TF* set to *sampling_TF* (10 on the *Xilinx ZU9EG*) is fine when the *MTS* is low, but that ends up lowering the *Efficiency* of *CBM* for *covariance* and *correlation* as the *MTS* increases in **Fig.5.8**, culminating with the case presented at *MTS* 1.43×. Since both tasks cannot be slowed down more than 1.4× by the accelerators, the higher *TF* applied by *CBM* as the *MTS* increases past their maximum slowdown reduces the problem caused by the fast traffic section changes (hence why the efficiency increases in the box plot past *MTS* 1.43×).

This lower efficiency was partially mitigated by the various improvements implemented in Section 5.2, however that was not enough. There are more ways to fix this: first, both lowering or increasing the time needed for `do_big_TF_adjustment()` to measure the bandwidth of the accelerator and the CPU could help these cases. When it comes to lowering the bandwidth

measurement time, in our experiments we noticed that measuring the bandwidth more frequently than every 250 μs produces unwanted results when it comes to reading the hardware registers. Another problem with this approach is that lowering the bandwidth measurement time could lead to a higher overhead due to the online policy executing more often. As such, the more realistic solution could be to increase the bandwidth measurement time of `do_big_TF_adjustment()`. However, this has the consequence of lowering the responsiveness of the online policy when it comes to other tasks (due to the `do_big_TF_adjustment()` function taking longer to complete). Finally, another way to handle this could be to implement an arbitrary check for such cases. However, there are risks associated with introducing too specific fixes for special cases, like the potential for other tasks to exceed the *MTS* due to unexpected consequences of the required changes.
More importantly, if one looks at **Fig.5.8**, when excluding *covariance* and *correlation*, the next worst case is an *Efficiency* of 0.96, meaning that otherwise the policy is capable of matching the *MTS* very closely.

5.3.3 MTS guarantee - Speed

Next, we wanted to measure how responsive the online policy is. Modern programs typically have phases that are compute intensive and phases that are memory intensive. Whenever there is a phase change in the CPU task, the *TF* set by *CBM* needs to be adjusted to prevent either the task exceeding the *MTS* or the accelerators being throttled too much. This process is not instantaneous: it takes some time for the online policy to trigger `do_big_TF_adjustment()`, and for the new *TF* to be set. The higher the speed of updating the *TF*, the more the online policy is responsive to change. This translates to *CBM* being able to properly guarantee the *MTS* for more programs.
To test the responsiveness of *CBM*, we created a worst-case synthetic benchmark specifically built for this purpose. This benchmark is a combination of *CpuW*, *CpuRW*, and *CpuR*, which can periodically switch between them thanks to a tunable parameter. The intensity of the sections themselves is

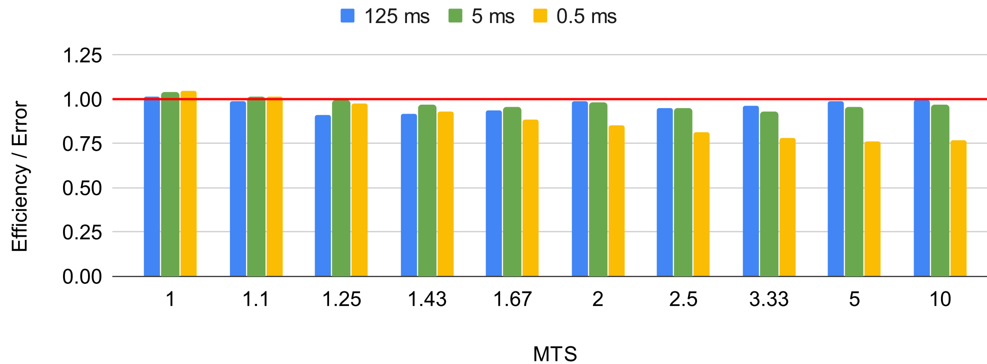


Figure 5.9: How *CBM* performs with a synthetic benchmark changing traffic type at different frequencies. In red, the target of 1.00 *Efficiency*. Ratios above 1.00 represent *Errors*. Ratios below 1.00 are called Efficiencies.

100%, meaning the benchmark is a task very subject to interference, and with different sections of it having very different bandwidth curves. Thus each section has a different *TF* for any wanted *MTS* (as previously demonstrated in Section 5.2.2.1). Then, we measured the slowdown that the benchmark was subject to for various wanted *MTSs* with *CBM* active, and with memory section changing frequencies every 0.5 ms, 5 ms and 125 ms. The frequency of 0.5 ms was explicitly chosen to stress the online policy, due to this value being close to the bandwidth measurement time of `do_big_TF_adjustment()` (0.3 ms). The frequency of 125 ms was chosen to have a case in which *CBM* should always be able to guarantee the *MTS*. Finally, the frequency of 5 ms was chosen as a case where we expect *CBM* to work fine, even if it is closer to the bandwidth measurement time.

Fig.5.9 presents the results of the experiment. The *Efficiency / Error* (vertical axis) is measured for the three changing frequencies (vertical bars) for various different *MTS* (horizontal axis, obtained by dividing 100 by multiples of 10). The red horizontal line is the target 1.00 *Efficiency*. What can be observed is that *CBM* is able to properly handle memory changes every 125 ms and 5 ms, as the *Efficiency* is particularly close to 1.00.

When the frequency becomes higher, at 0.5 ms, the online policy starts to

struggle, as expected. Resets are triggered very frequently due to the significant changes in traffic types. Just like for *covariance* and *correlation*, the online policy ends up spending too much time at *sampling_TF* (10). In this case, we measure a slight increase in execution time (i.e. the slight *Error*) when the *TF* should be lower than *sampling_TF* (observed for *MTS* 1.0× and 1.1×, where *TF* should be 3 and 7 respectively), as the accelerators are throttled too little (meaning their *TF* is higher than expected). Instead, in the cases when the *TF* should be higher than *sampling_TF* (observed for all other *MTS*), the accelerators are throttled too much, and the execution time for the synthetic benchmark decreases more than it should (i.e. a lower *Efficiency*). This is an extreme example of what happened with *covariance* and *correlation*.

A way to forcibly improve tasks changing memory phases frequently could be to lower the bandwidth measurement time. However, as previously stated and just like when discussing possible improvements for *covariance* and *correlation*, *CBM* on the *Xilinx ZU9EG* is limited by the frequency with which the hardware registers can be read.

Chapter 6

HeSoC-wide gradient-based Bandwidth Manager

In Chapter 5, we present a bandwidth manager capable of throttling accelerators in HeSoCs to reduce the slowdown that a *critical task* executing on the CPU complex is subject to. In this chapter, we present an HeSoC-wide gradient-based bandwidth manager, capable of throttling both the accelerators and other CPU cores to reduce the slowdown of a *critical task*.

First, we present an initial memory interference exploration, combining both CPU-based memory interference (first exposed in Section 4) and accelerator-based memory interference (previously studied as part of Section 5). After that, we describe in detail how both the accelerators and the CPU cores are throttled in this new version of the bandwidth manager. Finally, we present results that show both how the new version of the bandwidth manager performs when trying to satisfy a *MTS* for a *critical task*, as well as how much the other CPU cores and the accelerators are throttled to achieve the wanted *MTS*.

6.1 Memory interference in HeSoCs - Accelerators and CPU cores

As explained in Section 2.3, there are multiple sources of memory interference in modern HeSoCs, and they can combine with one another. In Section 4, we present a detailed analysis of how CPU cores can cause interference to one another both at the main memory level and at the shared cache level. In Section 5, we show how accelerators can cause interference to CPU tasks at the main memory level. In this section, we analyze how much interference (slowdown) a *critical task* can be subject to when both other CPU cores and accelerators are accessing the memory subsystem. Our experiments are executed on the *Xilinx ZU9EG* platform, with 4 CPU cores (one of which will be executing the *task under test* for which we measure the slowdown) and the ability to run 3 distinct accelerator workloads that access the main memory (i.e. executing 3 AccRW at the same time). In particular, we focus on two interesting cases, all with the same traffic type, stemming from the previous analysis done in Section 4:

1. CpuR vs. 3 CpuR + 3 AccRW: A task that accesses the main memory when executed in isolation is subject to interference by other CUs also accessing the main memory.

2. *PartitionedChaser* vs. 3 *PartitionedChaser* (eviction) + 3 AccRW: A task that does not accesses the main memory when executed in isolation is forced to access it because of cache evictions, and is subject to interference by other CUs also accessing the main memory.

The reported slowdowns are obtained by averaging the execution time of 1000 runs of the *task under test*, for varying *TF*s applied to both the accelerators and the CPU *interfering tasks*. In particular, we differentiate between AccTF for the *TF* applied to the accelerators, and CpuTF for the *TF* applied to the CPU *interfering tasks*.

Figure 6.1 shows the resulting slowdown for CpuR vs. 3 CpuR + 3 AccRW as AccTF grows on the horizontal axis. The curves represent different levels

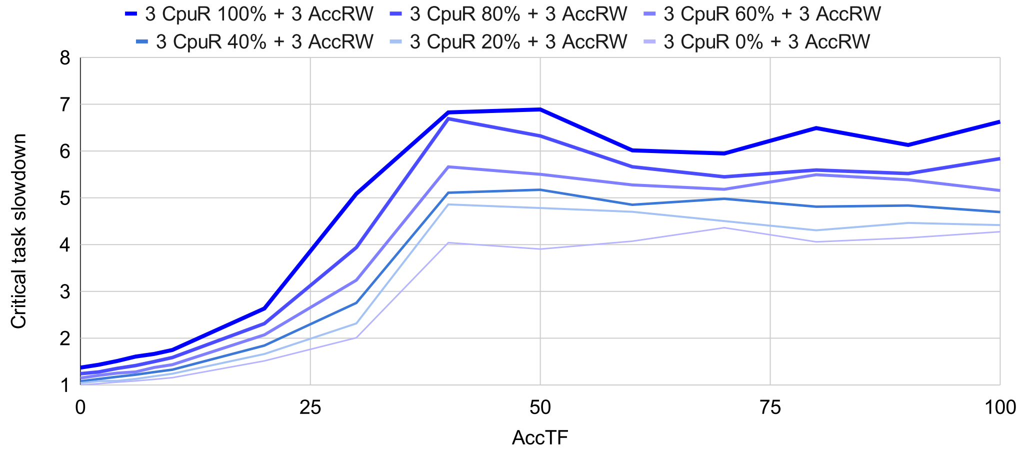


Figure 6.1: Slowdown of CpuR for varying degrees of interference by both the other CPU cores (running CpuR) and the accelerators.

of CpuTF for the *interfering tasks*, with thicker curves representing a higher CpuTF. As expected, the two interference types combine, slowing the *task under test* up to 7×. In particular, the plot shows that for this type of interference, AccTF is the factor that contributes the most to interference. This is because as the AccTF grows, the slowdown grows rapidly until it plateaus at AccTF 30, with a 4.2× slowdown even when CpuTF is 0. By comparison, with AccTF 0, the difference in slowdown between CpuTF 0 and CpuTF 100 is only 1.37×. As such, for cases like this, memory interference mitigation techniques should focus first and foremost on AccTF, and then on CpuTF.

Figure 6.2 shows the resulting slowdown for *PartitionedChaser* vs. 3 *PartitionedChaser* (eviction) + 3 AccRW as AccTF grows on the horizontal axis. In this case, while the two interference types do combine for a severely high slowdown (up to 25×), they do so in a different way than in Figure 6.1. In particular, the plot shows that increasing CpuTF contributes the most to interference. This is because with CpuTF 0, we observe absolutely no interference, even for high AccTF values. This is expected, as for low CpuTF, the *task under test* does not access the main memory. Once CpuTF is greater

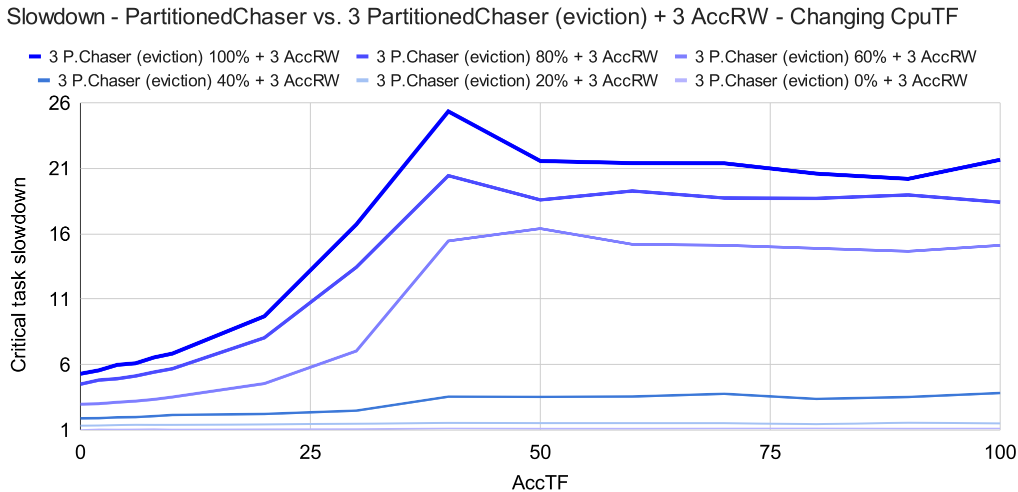


Figure 6.2: Slowdown of *PartitionedChaser* for varying degrees of interference by both the other CPU cores (running *PartitionedChaser* to cause evictions) and the accelerators.

than 40, slowdowns become severe, as the *task under test* is forced to access the main memory, and increasing AccTF starts to have an effect on experienced interference. While the actual contribution to the total interference is high for both CpuTF (5.2× at AccTF 0) and AccTF (4.2×, as already seen in Figure 6.1), memory interference mitigation techniques should focus first and foremost on keeping the CpuTF low, as that also prevents any interference from the accelerators.

In general, this section proves that: i) it is important to mitigate interference in HeSoCs, as when all the CUs use the memory subsystem at the same time, they can cause severe slowdowns to *critical tasks* (up to 25× on the *Xilinx ZU9EG*); ii) care should be taken in choosing which of the CUs to throttle, as different setups can lead to different CUs causing the highest amount of interference.

In the next section, we introduce a bandwidth manager capable of reducing interference even with high interference setups like the ones highlighted in this section.

6.2 Expanding the bandwidth manager

The bandwidth manager introduced in Section 5 is capable of throttling accelerators and can use that to satisfy *MTS* requests for a *critical task*. However, it is unable to throttle CPU cores, something that is extremely important, as shown in both Section 4 and Section 6.1. Not only that, but in Section 6.1 we prove how important it is to throttle the correct CUs to guarantee the *MTS* of a task in HeSoCs.

In this section, we present two changes to the bandwidth manager: i) we add to the bandwidth manager the ability to throttle other CPU cores; ii) we change how the algorithm used by the bandwidth manager decides to throttle the CUs, to accomodate the analysis we present in Section 6.1.

6.2.1 Bandwidth throttling the CPU cores

We define the budget of a core as the amount of bytes it is allowed to transfer within a specific time period.

Algorithm 11: Bandwidth throttling CPU cores.

```
1 cur_cpu_tf = x  // Set by other code
2 period = 100μs  // Default period
3 Function periodic_interrupt()
4     max_bytes_period = max_bw * period
5     budget = max_bytes_period * cur_cpu_tf / 100
6     program PMC to cause overflow interrupt if budget is exceeded
7     schedule next periodic interrupt after: period
8 Function overflow_interrupt()
9     busyWaitUntil(period)
```

To implement bandwidth throttling on the CPU cores, on each core executing the *interfering tasks*, we schedule a periodic interrupt. We show the involved functions in Algorithm 11. The periodic interrupt sets the budget of a core to the maximum amount of bytes it can transfer between consecutive calls of the periodic interrupt multiplied by CpuTF. After that, the periodic interrupt sets an overflow interrupt on a performance counter (PMC)

to trigger if more bytes are transferred than what has been allocated via the budget.

If the overflow interrupt is called the core enters a busy wait loop, until the next call of the periodic interrupt.

This setup ensures that a CPU core cannot transfer more bytes than what we allocate with CpuTF. While this does have the potential of leaving some bandwidth unused, the goal is, first and foremost, ensuring that the *MTS* of a *critical task* is satisfied. It will be up to the algorithm making use of CpuTF to find how high it can be set without the slowdown exceeding the *MTS*.

6.2.2 A gradient-based bandwidth manager

As established in Section 5, the online phase of *CBM* is based on two main functions: do_big_TF_adjustment and do_small_TF_adjustment. We start this expansion of *CBM* by focusing on do_small_TF_adjustment. We leave the changes to do_big_TF_adjustment as future work. In general, in the base form of *CBM*, do_small_TF_adjustment is capable of changing the *TF* such that the *critical task* bandwidth reaches the target bandwidth (corresponding to a *MTS*). The only problem is that do_small_TF_adjustment would be extremely slow at converging to the correct *TF*. This is the reason why do_big_TF_adjustment exists.

In Section 5 (Algorithm 7), do_small_TF_adjustment simply increases or decreases the *TF* according to the difference between the measured bandwidth and the target bandwidth. However, in the expanded version of the bandwidth manager, both AccTF and CpuTF may be changed to reach a target bandwidth. To understand which one should be changed, we use a gradient descent procedure with the goal of minimizing the difference between the bandwidth of the *critical task* and the target bandwidth.

Algorithm 13 is the new version of do_small_TF_adjustment. It takes as input the bandwidth measured just before the function is called (*No change*). First, it calls measure_tf_small_change (Algorithm 12) for both CpuTF and AccTF. This function changes the input *TF* by a little (starting with an

Algorithm 12: Auxillary functions for do_small_adj_gd.

```
1  Function measure_tf_small_change(tf_to_change, change, out_tf,
     out_critTask_BW, out_system_BW)
2      // Check how changing this TF affects the critTask_BW
3      base_tf = tf_to_change
4      repeat
5          tf_to_change += change
6          sleep(BW_measurement_period)
7          out_critTask_BW = measure_critTask_BW()
8          out_system_BW = measure_system_BW()
9          change_measured = base_critTask_BW >
             out_critTask_BW ? +1 : -1
10     until change_measured != change;
11     out_tf = tf_to_change
12     tf_to_change = base_tf
```

increase of 1 to the *TF* if the *No change* bandwidth is greater than the target bandwidth, or a decrease of 1 to the *TF* in all other cases), and measures the *critical task* bandwidth and the system bandwidth. To avoid issues with local *critical task* bandwidth minimums and maximums, it checks whether the obtained result matches with what one would expect (i.e. less bandwidth if the input *TF* has been increased, and more bandwidth if the input *TF* has been decreased). If it does not match, the measurement is done again, with a *TF* changed further.

Experimentally, we observe at most 4 consecutive measurements taken due to local bandwidth minimums. However, that is due to how the *TF* is altered by +/- 1. Higher changes may lower the amount of extra iterations needed. Once measure_tf_small_change has been called for both CpuTF and AccTF, do_small_adj_gd determines which of the measured bandwidths (*No change*, *CpuTF changed*, and *AccTF changed*) minimizes the difference to the target bandwidth. If that bandwidth corresponds to the case *AccTF changed*, AccTF is changed. If that bandwidth corresponds to the case *CpuTF changed*, CpuTF is changed.

Minimizing the distance to the target bandwidth ensures that the slowdown experienced by the *critical task* is *MTS*, regardless of what the other

Algorithm 13: Algorithm for the gradient-based version of do_small_TF_adjustment.

```
1  Function do_small_adj_gd()
2      change = base_critTask_BW > target_BW ? +1 : -1
3      // Check how changing the CPU TF affects the critTask_BW
4      measure_tf_small_change(cur_cpu_TF, change,
        cpu_tf_changed, critTask_BW_cpu_change,
        system_BW_cpu_change)
5      // Check how changing the Acc TF affects the critTask_BW
6      measure_tf_small_change(cur_acc_TF, change,
        acc_tf_changed, critTask_BW_acc_change,
        system_BW_acc_change)
7      // Get the distance from target_BW for all three measurements
8      BW_delta_base = abs(base_critTask_BW - target_BW)
9      BW_delta_cpu_change = abs(critTask_BW_cpu_change -
        target_BW)
10     BW_delta_acc_change = abs(critTask_BW_acc_change -
        target_BW)
11     // Did changing the Acc TF bring critTask_BW closer to target_BW?
12     // And was it the best?
13     if(BW_delta_acc_change < BW_delta_base) and
        (BW_delta_acc_change < BW_delta_cpu_change)
14         cur_acc_TF = acc_tf_changed
15     // Did changing the CPU TF bring critTask_BW closer to target_BW?
16     else if(BW_delta_cpu_change < BW_delta_base)
17         cur_cpu_TF = cpu_tf_changed
18     // Otherwise, did changing the Acc TF increase the system bandwidth?
19     else if(BW_delta_acc_change == BW_delta_base) and
        (system_BW_acc_change > base_system_BW)
20         cur_acc_TF = acc_tf_changed
21     // Did changing the CPU TF increase the system bandwidth?
22     else if(BW_delta_cpu_change == BW_delta_base) and
        (system_BW_cpu_change > base_system_BW)
23         cur_cpu_TF = cpu_tf_changed
```

Algorithm 14: CBM gradient-based algorithm.

```
1  cur_acc_TF = 0  // start with accelerators turned off
2  cur_cpu_TF = 0  // start with other CPU cores turned off
3  base_critTask_BW = 0
4  base_system_BW = 0
5  Function update_throttling_factor_gd()
6      base_critTask_BW = measure_critTask_BW()
7      base_system_BW = measure_system_BW()
8      do_small_adj_gd()
9  Function CBM_gradient_main()
10     sleep(BW_measurement_period)
11     isolation_BW = measure_critTask_BW()
12     target_BW = isolation_BW/MTS
13     while(true) {
14         update_throttling_factor_gd()
15         sleep(BW_measurement_period)
16     }
```

CUs are running to generate interference. After that, if there is no change, the algorithm checks if one between *CpuTF changed* and *AccTF changed* improves the system bandwidth utilization, prioritizing the one that does if such a case is present.

In Algorithm 14 we introduce a simplified main that periodically calls do_small_adj_gd.

6.3 Experimental evaluation

To evaluate the performance of do_small_adj_gd, we run the following tests: i) CpuR vs. 3 CpuR + 3 AccRW; ii) *PartitionedChaser* vs. 3 *Partitioned-Chaser* (eviction) + 3 AccRW. We decided to focus on these cases to ensure that the improved version of do_small_TF_adjustment is able to properly reach a target *MTS* for a given task, even in scenarios with high interference.

For each test, we run the *task under test* for 10× longer than normal to reduce the effects of do_small_adj_gd having to start from CpuTF 0 and

Results for:
CpuR vs. 3 CpuR + 3 AccRW

MTS	1.0	1.25	1.67	2.5	5.0
Slowdown	1.00	1.26	1.70	2.52	5.0
AccTF	0	15	21	57	100
CpuTF	0	2	2	1	44
BW DRAM (GB/s)	2.9	5.95	7.05	8.81	10.06

Table 6.1: Slowdown, final AccTF, final CpuTF and average system bandwidth towards the DRAM for CpuR vs. 3 CpuR + 3 AccRW. Notice how the final AccTF rapidly increases, while the final CpuTF does not.

Results for:
***PartitionedChaser* vs. 3 *PartitionedChaser* (eviction) + 3 AccRW**

MTS	1.0	1.25	1.67	2.5	5.0
Slowdown	1.00	1.26	1.71	2.53	4.97
AccTF	0	23	21	32	4
CpuTF	0	9	31	41	85
BW DRAM(GB/s)	0.4	7.22	7.94	9.83	6.54

Table 6.2: Slowdown, final AccTF, final CpuTF and average system bandwidth towards the DRAM for *PartitionedChaser* vs. 3 *PartitionedChaser* (eviction) + 3 AccRW. Notice how increasing the final CpuTF is prioritized, with the final AccTF having more variable values.

AccTF 0. We take measurements of the total *task under test* slowdown, the final AccTF, the final CpuTF and the average system bandwidth towards the DRAM, for key MTS values. Each test is repeated 10 times, and the average of each value is taken.

Table 6.1 shows the results for CpuR vs. 3 CpuR + 3 AccRW. The slowdown experienced by the task is very close to the *MTS*, for all the tests. To reach the target bandwidth as fast as possible, do_small_adj_gd prioritizes increasing AccTF in this case, as increasing CpuTF has a limited impact. This matches what was observed for Figure 6.1. Increasing the AccTF has the side effect of ensuring high system bandwidth towards the DRAM even for lower *MTS* values, as can be observed in the table.

Table 6.2 shows the results for *PartitionedChaser* vs. 3 *PartitionedChaser*

(eviction) + 3 AccRW. Even in this case, the slowdown experienced by the task is very close to the *MTS* for all the tests. As opposed to Table 4.2, CpuTF consistently grows as the *MTS* is increased. do_small_adj_gd prioritizes increasing CpuTF, since it increases the interference that the *task under test* is subject to faster than AccTF in the left region of Figure 6.2 (where do_small_adj_gd starts operating from, with AccTF 0 and CpuTF 0). The final AccTF presented in the table changes as a consequence of this initial increase in CpuTF. When the final CpuTF is low, raising the AccTF has a low impact, but it does increase the total system bandwidth, so it is raised by do_small_adj_gd. By contrast, when the final CpuTF is high (i.e. 80 in Figure 6.2), instead, raising AccTF has a very severe impact on the slowdown experienced by the *task under test*. This is the reason why with *MTS* 5.0 do_small_adj_gd sets a low final AccTF (hence a low total system bandwidth). While this behaviour of do_small_adj_gd does guarantee that the *MTS* request is satisfied for the *critical task*, having low total system bandwidth is suboptimal. Future improvements may include further prioritizing high system bandwidth utilization, to increase the system throughput.

In general, do_small_adj_gd is able to guarantee the proper *MTS* for a *task under test*. However, it is important to note that it is slow at converging, and that porting do_big_TF_adjustment to this improved version of *CBM* (e.g. by using an offline-based bandwidth characterization of notable workloads) is needed to provide a complete solution.

Chapter 7

Conclusions and Future work

This thesis tackles the issue of memory interference in Commercial Off-The-Shelf (COTS) Heterogeneous Systems-on-Chip (HeSoCs), which are used for a variety of applications in the industry. The presented work combines a detailed memory interference exploration with the design of a novel characterization-based mitigation technique, to achieve timing guarantees for *critical tasks* and system-wide throughput improvements.

7.1 Contribution Summary

This thesis brings three main contributions to the field. First, it challenges the assumption that main memory bandwidth saturation, caused by synthetic read-intensive workloads with 100% cache miss rate, represents the primary source of worst-case slowdowns among memory interference causes. Through a detailed evaluation on the *Xilinx ZU9EG* and the *Nvidia TX2* COTS HeSoCs, the research proves that platform-specific characteristics affect worst-case slowdowns, and that cache interference must also be taken into consideration. Within the same contribution, the thesis also catalogs the different types of interference that the memory subsystem may be subject to, highlighting an often underestimated type of cache interference: cache congestion.

Second, the thesis proposes *CBM*, a characterization-based bandwidth

manager for HeSoCs. It reduces the interference that a *critical task* is subject to down to a user-set tolerated level via an algorithm that bandwidth throttles the accelerators. Unlike prior work, *CBM* is capable of providing timing guarantees to workloads that change their usage of the memory subsystem at runtime, without requiring the tasks to be analyzed beforehand. It does so thanks to the offline characterization, which *CBM* uses to quickly and precisely adapt to any possible workload. *CBM* has an error rate of at most 2%, and employs an algorithm that improves system bandwidth utilization by keeping track of the slowdown of the *critical task*.

Finally, the thesis ends presenting an extension to *CBM* that also allows it to throttle CPU cores. In particular, we present an improvement to the logic of *CBM* that enables the bandwidth manager to guarantee slowdown requirements for *critical tasks* by throttling both accelerators and other CPU cores. This improvement has the ability to autonomously choose the regulation level for both accelerators and other CPU cores at runtime, depending on the workloads running on the CUs of the HeSoC. Results show that this technique works, meaning we present a working HeSoC-wide memory interference mitigation technique, albeit some improvements are still needed to make it viable for most workloads.

7.2 Limitations and Future Work

There are three key limitations that define the scope of future work.

First, while the extension to *CBM* presented in Chapter 6 does throttle tasks in HeSoCs, it is slow, and it is unable to adapt to workloads that change their usage of the memory subsystem at runtime. Such problems are not present int the version of *CBM* able to only throttle accelerators (Chapter 5), so it is a matter of adapting existing code and logic to the ability to also throttle CPU cores. To enable do_big_jump to choose the proper levels of CPU throttling, the first step should be a new and wider characterization of the target platform. We have also determined, by looking at the results presented in Chapter 4 (and some early experiments), that it should be possible to create an hierarchical version of do_big_jump that is capable

of understanding what type of interference a task is subject to (between main memory interference, cache conflict contention and cache congestion). With such information, do_big_jump should be able to determine the correct level of CPU and accelerator throttling extremely quickly, meaning the improved version of *CBM* would work even for very short tasks.

Second, the work presented in Chapter 5 and Chapter 6 is done exclusively on the *Xilinx ZU9EG*. One of the directions this work should be expanded in is targeting other HeSoCs. On the matter, we have done some early experiments on the *Nvidia AGX Orin* with a not-yet-published bandwidth throttler for the GPU that is compatible with *CBM*, and the mechanism works for simple tasks. However, further tests are needed.

Third, all versions of *CBM* only account for there to be only one *critical task*. A possible expansion of the work is to allow multiple *critical tasks*, with different priority levels.

7.3 Practical implications

The findings presented in this thesis have practical implications for multiple communities. For people trying to mitigate memory interference, the results presented in Chapter 4 highlight the need to do a proper characterization of the target platforms to find worst-case interference. The results also prove the existence of multiple types of interference, that certain mitigation techniques may not be able to handle, and which must be accounted for.

For people trying to use COTS HeSoC, *CBM* allows for a *critical task* to complete within a certain time while still having high throughput on the rest of the system, which can be especially useful for applications with strict SWaP requirements that need high performance computing.

7.4 Concluding remarks

Ultimately, this thesis tackled the issue of memory interference in COTS HeSoCs via detailed platform characterization and design of a novel mitiga-

tion technique. Rather than seeking a universal solution, this work proves the importance of understanding the characteristics of every platforms, and choosing how to mitigate interference accordingly. The three main contributions – detailed multi-type interference characterization, design of a characterization-based bandwidth manager via accelerator throttling, and further expansion of the bandwidth manager to also throttle CPU cores – are steps required for proper utilization of the hardware present in HeSoCs. These contributions are significant, but substantial work remains. As the number of CUs present in newer HeSoCs increases, reducing the effects of memory interference will become more and more important, like the findings we present in this thesis.

Bibliography

[1] M. Mattheeuws, B. Forsberg, A. Kurth, and 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.

[2] G. Brilli, A. Capotondi, P. Burgio, and A. Marongiu, "Understanding and mitigating memory interference in fpga-based hesocs," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 1335–1340.

[3] Y.-J. Lin, C.-L. Yang, J.-W. Huang, T.-J. Lin, C.-W. Hsueh, and N. Chang, "System-level performance and power optimization for mpsoc: A memory access-aware approach," *ACM Transactions on Embedded Computing Systems*, vol. 14, no. 1, 2015.

[4] H. Yun and P. K. Valsan, "Evaluating the isolation effect of cache partitioning on cots multicore platforms," *OSPERT 2015*, vol. 45, 2015.

[5] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–12.

[6] S. Mittal, "A survey of techniques for cache partitioning in multicore processors," *ACM Comput. Surv.*, vol. 50, no. 2, May 2017.

[7] A. T. Aurora Dugo, J.-B. Lefoul, F. G. De Magalhaes, D. Assal, and G. Nicolescu, "Cache locking content selection algorithms for arinc-653

compliant rtos," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 5s, 2019.

[8] A. M. Kaushik, M. Hassan, and H. Patel, "Designing predictable cache coherence protocols for multi-core real-time systems," *IEEE Transactions on Computers*, vol. 70, no. 12, p. 2098 2111, 2021.

[9] M. Hassan, A. M. Kaushik, and 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, p. 235 246.

[10] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 269–279.

[11] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *13th International Conference on Parallel Architecture and Compilation Techniques*, 01 2004, pp. 111–122.

[12] M. Bechtel and 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.

[13] A. Bansal, R. Tabish, G. Gracioli, R. Mancuso, R. Pellizzoni, and M. Caccamo, "Evaluating the memory subsystem of a configurable heterogeneous mpsoc," in *Proceedings of the Operating Systems Platforms for Embedded Real-Time applications*, 07 2018.

[14] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla, "On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, jan 2012. [Online]. Available: https://doi.org/10.1145/2086696.2086713

[15] J. Nowotsch and M. Paulitsch, “Leveraging multi-core computing architectures in avionics,” in *Ninth European Dependable Computing Conference*, 2012, pp. 132–143.

[16] N. Capodieci, R. Cavicchioli, I. S. Olmedo, M. Solieri, and 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.

[17] A. Serrano-Cases, J. M. Reina, J. Abella, E. Mezzetti, and F. J. Cazorla, “Leveraging hardware qos to control contention in the xilinx zynq ultrascale+ mpsoc,” in *Euromicro Conference on Real-Time Systems*, 2021.

[18] G. Brilli *et al.*, “Fine-grained qos control via tightly-coupled bandwidth monitoring and regulation for fpga-based heterogeneous socs,” in *60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.

[19] F. Restuccia *et al.*, “Is your bus arbiter really fair? restoring fairness in axi interconnects for fpga socs,” *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3358183

[20] Z. Jiang *et al.*, “Axi-ic-rt: Towards a real-time axi-interconnect for highly integrated socs,” *IEEE Transactions on Computers*, vol. PP, pp. 1–1, 01 2022.

[21] F. Restuccia *et al.*, “Axi hyperconnect: A predictable, hypervisor-level interconnect for hardware accelerators in fpga soc,” in *57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[22] R. Cavicchioli, N. Capodieci, M. Solieri, M. Bertogna, P. Valente, and A. Marongiu, “Evaluating controlled memory request injection to counter prem memory underutilization,” in *Job Scheduling Strategies for Parallel Processing*. Cham: Springer International Publishing, 2020, pp. 85–105.

[23] B. Forsberg, L. Benini, and A. Marongiu, "Heprem: A predictable execution model for gpu-based heterogeneous socs," *IEEE Transactions on Computers*, vol. 70, no. 1, pp. 17–29, 2021.

[24] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64.

[25] A. Saeed *et al.*, "Memory utilization-based dynamic bandwidth regulation for temporal isolation in multi-cores," in *IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 133–145.

[26] F. Farshchi *et al.*, "Bru: Bandwidth regulation unit for real-time multicore processors," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 364–375.

[27] H. Yun, W. Ali, S. Gondi, and S. Biswas, "Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms," *IEEE Transactions on Computers*, vol. 66, no. 7, pp. 1247–1252, 2017.

[28] M. Zini, D. Casini, and A. Biondi, "Analyzing arm's mpam from the perspective of time predictability," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 168–182, 2023.

[29] A. Zuepke *et al.*, "Mempol: polling-based microsecond-scale per-core memory bandwidth regulation," *Real-Time Systems*, vol. 60, pp. 369–412, 06 2024.

[30] G. Brilli, R. Cavicchioli, M. Solieri, P. Valente, and A. Marongiu, "Evaluating controlled memory request injection for efficient bandwidth utilization and predictable execution in heterogeneous socs," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 1, dec 2022. [Online]. Available: https://doi.org/10.1145/3548773

[31] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 145–154.

[32] J. Parkhurst, J. Darringer, and B. Grundmann, "From single core to multi-core: Preparing for a new exponential," in *2006 IEEE/ACM International Conference on Computer Aided Design*, 2006, pp. 67–72.

[33] H. G. Lee, N. Chang, U. Y. Ogras, and R. Marculescu, "On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, May 2008. [Online]. Available: https://doi.org/10.1145/1255456.1255460

[34] L. Benini and G. De Micheli, "Networks on chips: a new soc paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[35] ARM, *AMBA CHI Architecture Specification*. [Online]. Available: https://developer.arm.com/documentation/ihi0050/latest

[36] G. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proceedings Eighth International Symposium on High Performance Computer Architecture*, 2002, pp. 117–128.

[37] ARM, *Arm DynamIQ Shared Unit Technical Reference Manual - L3 cache partitioning*. [Online]. Available: https://developer.arm.com/documentation/100453/0401/L3-cache/L3-cache-partitioning

[38] Intel, *Improving Real-Time Performance by Utilizing Cache Allocation Technology*. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf

[39] N. Suzuki, H. Kim, D. de Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar, "Coordinated bank and cache coloring for temporal pro-

tection of memory accesses," in *IEEE 16th International Conference on Computational Science and Engineering*, 12 2013, pp. 685–692.

[40] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 89102. [Online]. Available: https://doi.org/10.1145/1519065.1519076

[41] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: A dynamic cache partitioning system using page coloring," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2014, pp. 381–392.

[42] AMD, *Zynq UltraScale+ Device Technical Reference Manual*, 2015. [Online]. Available: https://docs.amd.com/v/u/en-US/ug1085-zynq-ultrascale-trm

[43] NVIDIA, *TECHNICAL REFERENCE MANUAL - NVIDIA Parker Series SoC*, 2017. [Online]. Available: https://developer.nvidia.com/embedded/dlc/parker-series-trm

[44] ——, *NVIDIA Orin Series System-on-Chip - Technical Reference Manual*, 2022. [Online]. Available: https://developer.nvidia.com/downloads/orin-series-soc-technical-reference-manual

[45] Raspberry, *Processors - BCM2711*, 2019. [Online]. Available: https://www.raspberrypi.com/documentation/computers/processors.html#bcm2711

[46] ——, *Processors - BCM2712*, 2023. [Online]. Available: https://www.raspberrypi.com/documentation/computers/processors.html#bcm2712

[47] A. Ahmed and K. Skadron, "Hopscotch: a micro-benchmark suite for memory performance evaluation," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY,

USA: Association for Computing Machinery, 2019, p. 167172. [Online]. Available: https://doi.org/10.1145/3357526.3357574

[48] J. McCalpin, “Memory bandwidth and machine balance in high performance computers,” *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, 12 1995.

[49] L. Carletti and G. Brilli, *SynthMemBench*, 2025. [Online]. Available: https://github.com/LorenzoCarletti/SynthMemBench

[50] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, “Auto-tuning a high-level language targeted to gpu codes,” in *Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[51] K. Manev, A. Vaishnav, and D. Koch, “Unexpected diversity: Quantitative memory analysis for zynq ultrascale+ systems,” in *International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 179–187.

[52] J. Boudjadar, J. H. Kim, and 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, p. 105 114.

[53] M. Bechtel and H. Yun, “Memory-aware denial-of-service attacks on shared cache in multicore real-time systems,” *IEEE Transactions on Computers*, vol. 71, no. 9, pp. 2351–2357, 2022.

[54] P. K. Valsan, H. Yun, and 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.

[55] B. Forsberg, M. Solieri, M. Bertogna, L. Benini, and A. Marongiu, “The predictable execution model in practice: Compiling real applications for cots hardware,” *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 5, 2021.

[56] M. Hassan, A. Kaushik, and H. Patel, "Exposing implementation details of embedded dram memory controllers through latency-based analysis," *ACM Transactions on Embedded Computing Systems*, vol. 17, no. 5, 2018.

[57] M. M. Andreozzi, F. Antonio, L. Galli, G. Stea, and R. Zippo, "A milp approach to dram access worst-case analysis," *Computers & Operations Research*, vol. 143, 07 2022.

[58] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding and reducing memory interference in cots-based multi-core systems," *Real-Time Systems*, vol. 52, no. 3, p. 356 395, 2016.

[59] J. M. Aceituno, A. Guasque, P. Balbastre, J. Simó, and A. Crespo, "Hardware resources contention-aware scheduling of hard real-time multiprocessor systems," *Journal of Systems Architecture*, vol. 118, 2021.

[60] A. Alham implementation details of embedded DRAM memory controllers through latency-based analysismad and R. Pellizzoni, "Time-predictable execution of multithreaded applications on multicore systems," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2014.

[61] J. Martinez *et al.*, "Analytical characterization of end-to-end communication delays with logical execution time," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2244–2254, 2018.

[62] P. Sohal *et al.*, "Profile-driven memory bandwidth management for accelerators and cpus in qos-enabled platforms," *Real-Time Systems*, vol. 58, pp. 1–40, 09 2022.

[63] H. Aghilinasab *et al.*, "Dynamic memory bandwidth allocation for real-time gpu-based soc platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3348–3360, 2020.

[64] M. Zini *et al.*, "Profiling and controlling i/o-related memory contention in cots heterogeneous platforms," *Software: Practice and Experience*, vol. 52, no. 5, pp. 1095–1113, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3053

[65] J. McCalpin, *Sustainable Memory Bandwidth in Current High Performance Computers*, 1995. [Online]. Available: https://www.cs.virginia.edu/~mccalpin/papers/bandwidth/node2.html#SECTION00020000000000000000

[66] J. Barrera, L. Kosmidis, H. Tabani, E. Mezzetti, J. Abella, M. Fernandez, G. Bernat, and 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*, ser. SAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 580589. [Online]. Available: https://doi.org/10.1145/3341105.3373955

[67] "Repository with cache partitioning setup for Raspberry Pi 5." [Online]. Available: https://github.com/ColeStrickler/Pi5-CacheWayPartition

[68] AMD, *Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891)*, 2025. [Online]. Available: https://docs.amd.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview

[69] AMD, "AXI Performance Monitor v5.0," https://docs.amd.com/v/u/en-US/pg037_axi_perf_mon, 2017.

[70] Synopsys, "Enhancing Arm SoCs Performance with Smart Monitors," https://www.synopsys.com/blogs/chip-design/smart-monitors-improve-arm-socs-performance.html, 2023.

[71] G. Valente, G. Brilli, T. D. Mascio, A. Capotondi, P. Burgio, P. Valente, and A. Marongiu, "Fine-grained qos control via tightly-coupled bandwidth monitoring and regulation for fpga-based heterogeneous socs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 36, no. 2, pp. 326–340, 2025.