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

Demystifying Memory Interference: Analysis and Mitigation of the Memory Contention Problem in Heterogeneous Systems-on-Chip

Candidato	Dott. Andrea Serafini
Relatore (Tutor):	Prof. Paolo Valente
Correlatore (Co-Tutor):	Prof. Andrea Marongiu
Relatore Aziendale:	Dott. Alessandro Biasci
Coordinatore del Corso di Dottorato:	Prof. Andrea Marongiu

Abstract

Modern embedded Systems-on-Chip (SoCs) integrate a large number of parallel processing units – such as multicore CPUs and hardware accelerators – that share common hardware resources. While this high level of parallelism enhances performance and energy efficiency, it also introduces contention, as multiple compute units compete for shared resources. Shared memory, in particular, becomes a major bottleneck due to its limited bandwidth, leading to severe contention that degrades performance and undermines predictability. These effects are especially problematic in mixed-critical and real-time systems, where bounded latency and timing guarantees are essential. Understanding the sources of memory interference and developing effective mitigation mechanisms are therefore key to improving isolation and predictability.

This thesis tackles the problem of memory bandwidth contention in multicore and heterogeneous embedded platforms by combining a detailed analysis of interference phenomena with the design of novel mitigation strategies. It begins with a systematic characterization of memory contention in multicore platforms, analyzing how interference arises across different levels of the shared memory hierarchy—including DRAM, caches, and microarchitectural components. Experiments on Xilinx UltraScale+ and NVIDIA Xavier AGX platforms reveal how interference at each level impacts performance differently, highlighting the main contention points that limit scalability.

Building on these insights, the thesis examines Memory Bandwidth Management Schemes (MBMSs) – software mechanisms designed to mitigate memory interference among parallel compute units. In dynamic mixed-criticality systems, where tasks frequently alternate across units, MBMSs mitigate interference by periodically reconfiguring memory bandwidth thresholds and regulating access accordingly. The effectiveness of an MBMS strongly depends on the granularity of reconfiguration, a factor often overlooked in prior work. This thesis provides an in-depth analysis of the bandwidth reconfiguration process, comparing two common approaches: synchronous and

asynchronous. Experiments on a Xilinx UltraScale+ platform show that asynchronous reconfiguration significantly improves system responsiveness over the synchronous method.

The analysis is then extended to heterogeneous platforms from NVIDIA, integrating GPUs as primary hardware accelerators. This thesis proposes a novel MBMS to protect CPU clusters from GPU-induced memory interference. The approach leverages CUDA Green Contexts – a built-in NVIDIA feature – making it applicable across all supported architectures and enabling fully software-driven, dynamic control of GPU memory bandwidth. Experiments on the two latest platforms by NVIDIA – the AGX Orin and the AGX Thor – demonstrate that this method effectively reduces GPU interference on CPU cores.

Overall, this thesis advances the understanding of memory interference phenomena in both homogeneous and heterogeneous SoCs and proposes adaptive bandwidth control techniques that enhance performance predictability. The results provide a foundation for designing more efficient and QoS-aware embedded architectures through coordinated hardware-software memory management.

Acknowledgments

Ten days before completing this journey, I realize how crucial the support of so many people has been in reaching this milestone. Whether through their time, a shared laugh, or technical expertise, countless individuals have been essential to my path. I want to express my deepest gratitude to them all.

First, I would like to thank those who welcomed and guided me: my academic supervisors Paolo Valente and Andrea Marongiu, and my industry mentors Alessandro Biasci and Claudio Scordino. Their expertise and experience shaped my research, providing not just invaluable technical advice but also genuine friendship and encouragement that made this scientific adventure truly memorable. I also want to recognize two key figures who launched my journey, Paolo Gai and Riccardo Schiavi – they gave me this opportunity and shared both insightful guidance and heartfelt moments.

Second, I would like to thank my colleagues at Evidence S.R.L and the University of Modena and Reggio Emilia. The endless technical discussions and coffee breaks (sometimes ridiculously long) were vital for building my skills and forming lasting friendships. Among them, I would like to thank my PhD peers: Francesco Cosimi, Gerlando Sciangula, Andrea Stevanato, Raffaele Giannessi, Davide Bellassai, Lorenzo Carletti, Federico Motta, and Andrea Artioli. Working alongside them was a true privilege – our relationships went far beyond professional advice, and I am honored to call them friends today.

Third, I would like to thank my friends, both old and new. There are too many to name here, but one deserves special mention: Simone, who has been my confidant these past few years. Together with everyone, their lighthearted moments helped me grow while keeping life and work in balance. I hope I have been as meaningful to them as they have been to me.

Finally – and most importantly – my deepest gratitude goes to my family: my brother Luca, parents Paola and Massimo, and grandparents Gilda and Enrico. They were my safe haven through constant travel between Pavullo, Modena, and Pisa. They listened patiently, endured my challenges and encouraged me, even when the path ahead felt daunting. A special nod to

Luca, whose words—or often just a look—always conveyed exactly what he thought. To him, I owe the most significant part of this achievement.

Ringraziamenti

A dieci giorni dalla conclusione di questo percorso, mi rendo conto che senza l'aiuto di tante persone probabilmente non sarei arrivato a questo traguardo. Chi con il proprio tempo, chi con una risata, chi con le proprie competenze tecniche, tante persone sono state un punto di riferimento fondamentale in questo percorso. Per questo, voglio esprimere la mia gratitudine.

Il primo ringraziamento va a chi in questo percorso mi ha accolto e guidato: i miei supervisori accademici, Paolo Valente e Andrea Marongiu, e i miei supervisori in azienda, Alessandro Biasci e Claudio Scordino. Con le loro competenze e la loro esperienza mi hanno guidato nel mio percorso di ricerca, offrendo non solo preziosi consigli tecnici ma anche amicizia e un supporto umano che hanno reso questo viaggio scientifico indimenticabile. A loro mi sento di aggiungere altre due figure fondamentali per il mio percorso, Paolo Gai e Riccardo Schiavi. Essi per primi mi hanno dato l'opportunità di iniziare questo percorso in azienda, e hanno condiviso con me consigli preziosi e risate sincere.

Il secondo ringraziamento va ai miei colleghi, quelli di Evidence S.R.L e quelli dell'Università di Modena e Reggio Emilia. Con loro, il continuo confronto tecnico e le lunghe pause caffè (a volte lunghissime) sono stati fondamentali per accumulare competenze e creare amicizie. Tra loro voglio ringraziare in particolare i miei colleghi di dottorato: Francesco Cosimi, Gerlando Sciangula, Andrea Stevanato, Raffaele Giannessi, Davide Bellassai, Lorenzo Carletti, Federico Motta e Andrea Artioli. Lavorare con loro è stato un privilegio. Con questi compagni di viaggio, il rapporto umano è andato ben oltre il consiglio tecnico, e oggi ho l'onore di poterli chiamare amici.

Il terzo ringraziamento va ai miei amici, quelli di sempre ma anche quelli nuovi. Ho la fortuna di poter dire che sono tanti, troppi per poterli elencare tutti qui. Tuttavia, vorrei ringraziarne uno in particolare, Simone, che si è rivelato essere il mio *caro diario* in questi ultimi anni. Con lui e con tutti loro, i momenti di leggerezza sono stati fondamentali per crescere e per mantenere in equilibrio vita e lavoro. Spero di essere importante per loro tanto quanto loro lo sono stati per me in questi ultimi anni.

L'ultimo ringraziamento, ma il primo per importanza, va alla mia famiglia: mio fratello Luca, i miei genitori Paola e Massimo, e i miei nonni Gilda ed Enrico. A loro modo, essi sono stati il porto sicuro nel mio continuo viaggiare tra Pavullo, Modena e Pisa. Mi hanno ascoltato, sopportato e spronato continuamente nel cercare di rincorrere questo traguardo, anche nei momenti in cui la strada per Pisa mi sembrava più in salita. Tra loro, un ringraziamento particolare va a mio fratello Luca, che spesso con una parola, e ancora più spesso con un solo sguardo, mi fa capire quello che pensa. È a lui che devo la parte più importante di questo percorso.

Contents

1 Introduction **1**

1.1 The Memory Interference Problem 3

1.1.1 Open Challenge: Hardware Characterization 3

1.1.2 Open Challenge: Memory Interference Mitigation in Mixed-Criticality Systems 4

1.2 Contributions 4

1.2.1 Memory Interference Characterization 5

1.2.2 Memory Bandwidth Reconfiguration Analysis 5

1.2.3 Memory Bandwidth Management Schemes in Heterogeneous Platforms 6

1.3 Thesis Organization 7

2 Background: COTS Heterogeneous Systems on Chip **10**

2.1 Towards heterogeneous systems 10

2.2 Architectural Model 11

2.2.1 Host complex 12

2.2.2 Accelerator complex 12

2.2.3 Memory subsystem 13

2.3 Relevant HeSoCs 15

2.3.1 Xilinx UltraScale+ 15

2.3.2 Xilinx Versal ACAP 16

2.3.3 NVIDIA Jetson TX2 17

2.3.4 NVIDIA Jetson Xavier 18

2.3.5 NVIDIA Jetson AGX Orin 19

2.3.6 NVIDIA Jetson AGX Thor 20

3 Related Work **22**

3.1 Memory Interference Characterization 22

3.1.1 Shared Cache Level 23

3.1.2 System interconnect and Main Memory Level 23

3.2 Memory Interference Mitigation 25

3.2.1 Shared Cache Interference 26

3.2.2 System Interconnect and Main Memory Interference within the Host Complex 27

3.2.3 System Interconnect and Main Memory interference between the *Host-Accelerator* Complexes 30

4 Taking a Closer Look at Host-Level Memory Interference Effects **33**

4.1 Background 34

4.1.1 Architectural Model 34

4.1.2 Cache Memory 36

4.1.3 Workloads 38

4.2 Evaluation 43

4.2.1 Main Memory Bandwidth Saturation 43

4.2.2 Memory Interference Effects Analysis 44

4.2.3 Last Level Cache thrashing 50

4.3 Discussion and Conclusion 59

5 Synchronous vs. Asynchronous Reconfiguration of Memory Bandwidth in MBMSs: a Comparative Analysis **61**

5.1 Hardware and Software Model 63

5.2 Memory Bandwidth Management Schemes 65

5.2.1 Bandwidth Regulation 65

5.2.2 *SYNCHRONOUS* 66

5.2.3 *ASYNCHRONOUS* 68

5.2.4 Discussion 70

5.3 Synthetic benchmark 70

5.3.1 Main task . . . 70
5.3.2 Other tasks . . . 71
5.3.3 Output . . . 73
5.3.4 Running configurations . . . 73
5.4 Evaluation Setup . . . 74
5.4.1 Hardware and software setup . . . 74
5.4.2 QoS slowdown threshold . . . 75
5.4.3 *SYNCHRONOUS period* parameter configuration . . . 75
5.5 Synthetic Benchmark Evaluation . . . 76
5.5.1 Evaluation Process . . . 76
5.5.2 Synthetic Benchmark Configuration . . . 77
5.5.3 Evaluation Results . . . 78
5.6 Real-World Benchmarks Evaluation . . . 83
5.6.1 Evaluation Setup . . . 83
5.6.2 Evaluation results . . . 85
5.7 Discussion . . . 87
5.8 Conclusion . . . 89

6 Mitigating GPU-to-CPU Memory Interference through Spatial GPU Throttling and Cuda Green Contexts 90
6.1 Background . . . 92
6.1.1 Hardware and Software Model . . . 92
6.1.2 Graphic Processing Units . . . 94
6.2 *Spatial*-domain based MBMS . . . 100
6.2.1 Design . . . 100
6.2.2 Implementation . . . 102
6.3 Synthetic Benchmarking . . . 105
6.3.1 *cpu_synth* . . . 105
6.3.2 GPU Memory Traffic Generators . . . 106
6.4 Synthetic Benchmark Evaluation . . . 112
6.4.1 Evaluation Setup . . . 112
6.4.2 GPU Memory Traffic Generators Configurations . . . 113
6.4.3 Evaluation Process . . . 114

6.4.4 Evaluation Results . . . 114
6.5 Real-world benchmarks evaluation . . . 122
6.5.1 Evaluation Setup . . . 122
6.5.2 Evaluation Results . . . 124
6.6 Conclusion . . . 132

7 Conclusions and Future Work **134**
7.1 Contribution Summary . . . 134
7.2 Limitations and Future Work . . . 136
7.3 Practical Implications . . . 137
7.4 Concluding Remarks . . . 137

List of Figures

1.1 HeSoC parallel topology. 2

2.1 HeSoC Architecture Model. 11
2.2 Architecture of Xilinx ZCU102 development platform. 15
2.3 Architecture of Xilinx VCK190 development platform. 16
2.4 Architecture of NVIDIA TX2. 17
2.5 Architecture of NVIDIA AGX Xavier. 18
2.6 Architecture of NVIDIA AGX Orin. 19
2.7 Architecture of NVIDIA AGX Thor. 20

3.1 Categorization of memory interference mitigation techniques. . 25

4.1 *Nvidia TX2* architectural model - *Host*-complex only. 35
4.2 *Xilinx ZU9EG* architectural model - *Host*-complex only. . . . 35
4.3 Cache Memory Architecture. 36
4.4 Cache Operation Diagram. 37
4.5 DRAM bandwidth reached by the three traffic types on our reference SoCs. 45
4.6 Example of slowdown plot including both the synthetic and PolyBench-acc benchmarks. 46
4.7 ***Nvidia TX2* (a,c,e), *Xilinx ZU9EG* (b,d,f)**. Time increase for Synthetic and Polybench benchmarks. Note: each plot has a different maximum y axis value. 47
4.8 Slowdown and LLC refills caused by Cache Thrashing and Main Memory Interference on ***Nvidia TX2*** 52

4.9 Slowdown and LLC refills caused by Cache Thrashing and Main Memory Interference on ***Xilinx ZU9EG*** 53
4.10 ***Nvidia TX2*** **(a,c,e),** ***Xilinx ZU9EG*** **(b,d,f)**. LLC Write-backs caused by Cache Trashing and Main Memory Interference 55
4.11 *READ_MISS* (*fp* = 512KB) interfered by *MEMSET*. The *CPU STALLS because of load miss* (Subplot 4.11a) are measured on the *task under test*. The *WRITE OPERATIONS that stall the pipeline because the store buffer is full* (Subplot 4.11b) are measured on the *interfering tasks* 58

5.1 *Xilinx ZU9EG* architecture. 63
5.2 Execution model of the **synchronous** *reconfiguration* technique. 66
5.3 Execution model of the **asynchronous** *reconfiguration* technique. 68
5.4 Slowdown induced by the overhead of *SYNCHRONOUS* as a function of its *period* configuration. 75
5.5 *MCS SLOWDOWN* results obtained using the synthetic benchmark. 79
5.6 *BANDWIDTH PERCENTAGE* perceived by *other* tasks *inside* and *outside* the MCSs. 81
5.7 *MCSs SLOWDOWN* and *BANDWIDTH PERCENTAGE* inside and *outside* MCSs obtained using the *PolyBench* tasks. . 85

6.1 Architecture of *NVIDIA AGX Orin* development platform. . . 93
6.2 Architecture *NVIDIA AGX Thor* development platform. . . . 94
6.3 GPU architecture. 95
6.4 CUDA Threads hierarchical organization. 98
6.5 CUDA Offload Time diagram. 98
6.6 GPU *spatial*-domain regulation overview 101
6.7 GPU *spatial*-domain regulation implementation. 102
6.8 GPU SM Partitioning 104
6.9 Synthetic benchmark results on *NVIDIA AGX Orin*, *coalesced* 116
6.10 Synthetic benchmark results on *NVIDIA AGX Thor*, *coalesced* 117
6.11 Synthetic benchmark results on *NVIDIA AGX Orin*, *strided* . 120

6.12 Synthetic benchmark results on the *NVIDIA AGX Thor*, *strided* 121
6.13 GPU slowdown results obtained using PolyBench-ACC benchmarks on the *NVIDIA AGX Orin* platform. 126
6.14 GPU slowdown results obtained using PolyBench-ACC benchmarks on the *NVIDIA AGX Orin* platform. 127
6.15 CPU slowdown results obtained using PolyBench-ACC benchmarks on the *NVIDIA AGX Thor* platform. 129
6.16 GPU slowdown results obtained using PolyBench-ACC benchmarks on the *NVIDIA AGX Thor* platform. 130

List of Tables

4.1 Relevant hardware information for our reference platforms. . . 36

Chapter 1

Introduction

Today, physical and industrial processes are increasingly being automated using *Cyber-Physical Systems* (CPSs): integrated computing systems that combine perception and actuation to interact with the physical world [76]. Representative examples of CPSs include autonomous vehicles, aerial drones, and industrial robots. Such systems seamlessly weave computational intelligence into physical processes, delivering both high-performance and consistent reliability. However, designing CPSs introduces fundamental challenges, as the hardware employed for such systems must meet stringent performance requirements and physical constraints [75]. These systems often need to execute computationally intensive workloads – such as computer vision or signal processing tasks for autonomous vehicles – while meeting rigorous timeliness requirements. At the same time, they have to operate within severe *SWaP* constraints (Size, Weight, and Power) imposed by the embedded deployment scenarios. The tension between computational requirements and physical limitations restricts the available hardware platform options to deploy CPSs, representing one of the central challenges in CPS design.

To address the demanding constraints inherent in CPSs, the embedded systems industry has increasingly adopted Commercial Off-The-Shelf (COTS) Heterogeneous Systems-on-Chip (HeSoCs) as the primary hardware platform for CPS deployment [25, 27]. COTS HeSoCs are cheap, mass-produced, general-purpose hardware platforms that integrate heterogeneous

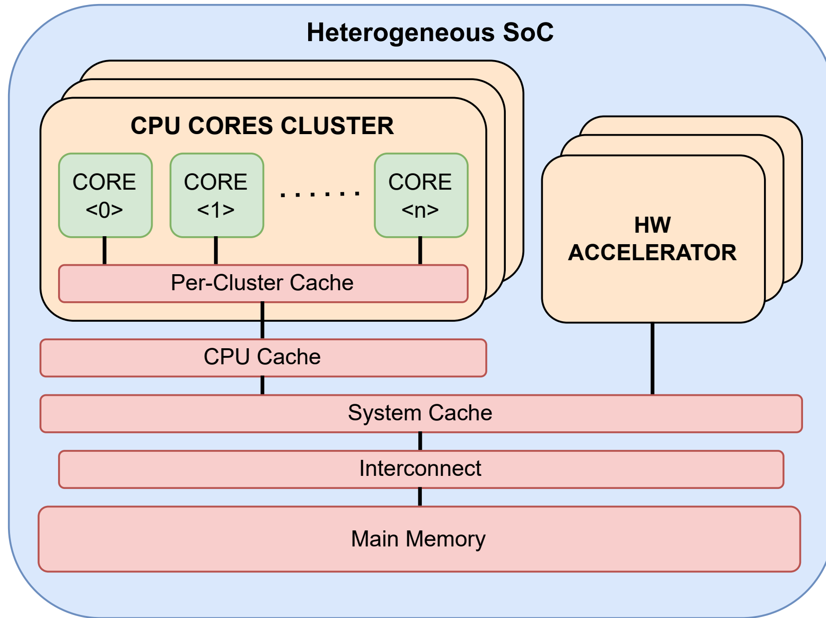


Figure 1.1: HeSoC parallel topology.

compute units – such as multicore CPUs and specialized accelerators – onto a single chip. Compared to custom Application-Specific Integrated Circuits (ASICs), COTS HeSoCs offer more flexibility and substantial economic advantages [35]. Moreover, the parallelism inherent in these integrated compute units enables systems to process multiple tasks simultaneously, delivering high levels of performance within a compact form factor [48]. This combination of affordability, flexibility, high performance, and limited size has established COTS HeSoCs as the industry standard for CPS deployment.

Modern COTS HeSoCs are built following the parallel topology shown in Figure 1.1, based on multiple parallel compute units (CPU cores and hardware accelerators) sharing certain common resources [11]. This topology exposes both the power and the limits of COTS HeSoCs: on the one hand, the use of shared resources facilitates integration of different compute units, enhancing performance and scalability [74]; on the other hand, it introduces contention, as multiple compute units may compete for said shared resources [45, 123].

1.1 The Memory Interference Problem

In modern HeSoCs, the memory subsystem is deeply shared across compute units. Consequently, simultaneous access demands from multiple units can overload the subsystem, yielding non-uniform memory access latencies [45]. This contention manifests distinctly at multiple levels of the memory hierarchy: at the shared cache level, the limited capacity cannot accommodate data for all CPU cores in a multicore CPU, forcing cores to access the main memory [117]; meanwhile, the main memory may not have sufficient bandwidth to serve requests from all compute units concurrently [123]. In those high-load scenarios, the shared memory hierarchy becomes a major hardware bottleneck of HeSoC platforms, causing the tasks scheduled on parallel compute units to suffer slowdown due to latency increase [20, 70, 14]. So far, this has been the principal obstacle to the adoption of HeSoCs in the context of CPSs.

Over the past two decades, the research community has extensively investigated the problem of memory interference [81, 80]. Several works in the literature provide characterizations of memory interference patterns [20, 109]; other works provide software- or hardware-based techniques to mitigate interference [124, 130, 104]. These works have contributed to the understanding of memory interference and have demonstrated the feasibility of various mitigation approaches. However, significant open challenges remain.

1.1.1 Open Challenge: Hardware Characterization

From a hardware perspective, contemporary embedded HeSoCs exhibit increasing architectural complexity [60]. Modern platforms integrate increasingly capable heterogeneous accelerators – such as GPUs and FPGAs – together with always deeper and more complex memory hierarchies and memory controllers. This architectural trend creates platform-specific memory interference patterns that are difficult to predict and generalize across different HeSoC designs [115, 32, 120].

Understanding interference on each platform requires a deep per-platform analysis to identify critical bottlenecks that fundamentally limit system scala-

bility. Moreover, to design effective mitigation strategies, we need techniques that comprehensively account for these sophisticated hardware configurations, including all integrated accelerators and their unique memory access characteristics.

1.1.2 Open Challenge: Memory Interference Mitigation in Mixed-Criticality Systems

Modern CPSs typically exhibit *mixed-criticality* execution paradigms and high levels of *dynamism* [78, 41]. In such systems, high-priority (*critical*) tasks must co-exist – and frequently alternate – with low-priority (*best-effort*) tasks. *Critical* tasks, in particular, must respect strict timing constraints (*deadlines*) by which their execution must be completed. The concurrent execution of tasks with conflicting requirements jeopardizes the timing guarantees of *critical* tasks, since memory interference from *best-effort* tasks can introduce unpredictable delays [70]. Even worse, this problem is further exacerbated when *critical* and/or *best-effort* tasks rely on hardware accelerators: the unique memory access pattern of these compute units significantly increases the level of interference within the system [36].

Addressing these issues requires mitigation strategies that can adapt dynamically to highly heterogeneous and complex workload patterns, while ensuring temporal isolation despite the competing demands of co-executing tasks.

1.2 Contributions

This thesis tackles the problem of memory interference in HeSoCs by combining a detailed characterization of interference phenomena with the analysis and the design of novel mitigation strategies.

1.2.1 Memory Interference Characterization

First, we focus on the memory interference that occurs at all levels of the shared memory hierarchy of modern HeSoCs. Rather than isolating specific hierarchy levels – as prior work typically does [115, 109, 62] – this study presents an integrated view of multiple memory interference causes: main memory bandwidth saturation, cache *thrashing*, and micro-architectural phenomena within CPU cores. By jointly analyzing these factors, we systematically decompose the overall interference into the individual contributions arising at each level of the memory hierarchy, *demystifying* how each component shapes the global interference effect – something that cannot be obtained when studying each cause in isolation.

In particular, we expose how the memory interference generated by CPU cores at each level of the memory hierarchy contributes to the global memory interference. Experiments on widely adopted COTS platforms such as Xilinx UltraScale+ and NVIDIA Tegra TX2 reveal how different types (i.e., at different hierarchy levels) of memory interference impact performance differently, highlighting the main contention points that limit scalability.

This study also challenges a widespread assumption in the literature: that main memory bandwidth saturation, induced by synthetic read-intensive workloads with 100% cache miss rates, represents the primary source of worst-case slowdowns among memory interference causes [83, 5, 98, 70]. Extensive experiments demonstrate that this assumption does not universally hold, as other causes of memory interference may cause worse slowdowns than main memory bandwidth saturation.

1.2.2 Memory Bandwidth Reconfiguration Analysis

Building on these insights, this work then focuses on *Memory Bandwidth Management Schemes* (MBMSs), software mechanisms designed to mitigate the effects of main memory interference in mixed-criticality systems [80]. In such systems – where uncontrolled memory interference can cause *critical* tasks to miss *deadlines* – MBMSs limit memory bandwidth allocation of *best-effort* tasks, thereby enabling *critical* tasks to access memory undis-

turbed [124]. *Dynamic mixed-criticality* systems, in particular, present an additional challenge: as tasks with varying criticality frequently enter and exit the system, MBMSs must periodically reconfigure bandwidth thresholds to maintain interference control. As a result, the effectiveness of a MBMS critically depends on both the quality of its *regulation* technique and the timeliness of its *reconfiguration*: *reconfiguration* techniques that cannot meet the time granularity at which tasks alternate in the system lead to wrong bandwidth threshold assignments and, in turn, wrong bandwidth regulation scenarios.

Although prior work has extensively investigated bandwidth *regulation* techniques [124, 130, 105], the *reconfiguration* component of MBMSs has received limited attention [51, 121]. This study presents the first in-depth analysis of bandwidth reconfiguration on HeSoC, comparing two fundamental implementation techniques, *synchronous* and *asynchronous*. The *synchronous* technique relies on periodic checks, providing fixed and predictable latency. In contrast, the *asynchronous* technique uses task-triggered interrupts, offering potentially lower latency and overhead. We evaluate these techniques on a Xilinx Zynq Ultrascale+ platform using both synthetic and real-world benchmarks (PolyBench suite). The results show that the *asynchronous* technique improves control granularity by up to 19× compared to the *synchronous* technique, reducing response times from millisecond to microsecond scale.

1.2.3 Memory Bandwidth Management Schemes in Heterogeneous Platforms

Lastly, the study on MBMSs is extended to the heterogeneous accelerators. Specifically, we focus on COTS HeSoCs provided by NVIDIA, integrating *Graphic Processing Units* (GPUs) as the main hardware accelerators. GPUs are massively parallel accelerators that deliver teraflops of compute throughput, enabling efficient execution of data-parallel workloads. In particular, in NVIDIA HeSoCs, GPUs share – and compete for – the main memory with CPU cores, posing critical challenges for predictable CPU task execution.

In this thesis, we propose a novel MBMS specifically designed to protect CPU execution from GPU-induced memory interference. Compounding previous approaches, which achieve CPU isolation through GPU idling [37] or by exploiting memory controller hardware throttling mechanisms [104], our MBMS operates in the spatial domain: it dynamically limits the number of streaming multiprocessors (SMs) available to GPU kernels based on CPU workload criticality. This spatial partitioning approach leverages *CUDA Green Contexts* – a NVIDIA-native technology [58] – to enable portable, transparent, and runtime-adaptive control of GPU resource consumption, ensuring predictable CPU performance while allowing efficient concurrent execution.

Extensive evaluation performed on latest available NVIDIA HeSoCs – NVIDIA AGX Orin and AGX Thor – shows that this approach effectively reduces GPU-induced slowdowns on CPU cores across diverse workload configurations. Moreover, this analysis also uncovers a pathological case on the NVIDIA AGX Thor platform: even a single SM can generate GPU memory access patterns (specifically high-stride write operations) that can saturate the memory controller, inducing up to 5× slowdown on CPU cores. This finding highlights the limits of spatial domain regulation and demonstrates that platform-specific phenomena require adaptive mitigation strategies.

Collectively, these contributions establish a foundation for designing predictable systems through coordinated hardware-software memory management. The insights derived from interference characterization, reconfiguration analysis, and heterogeneous platform management enable more effective and adaptive memory interference control techniques, advancing the state of the art in CPSs design.

1.3 Thesis Organization

This thesis is organized around three complementary research tracks, each addressing a distinct aspect of memory interference in COTS HeSoCs. The following chapters present the work in a logical progression from foundational characterization through mitigation analysis to practical system design.

Chapter 2 provides essential background on COTS HeSoCs, their architectural models, and examples of relevant embedded platforms employed to implement CPSs. This chapter establishes the terminology and architectural foundation necessary to understand memory interference phenomena in heterogeneous systems.

Chapter 3 synthesizes prior work on memory interference characterization and mitigation, organizing the literature into coherent categories based on hierarchy level (shared cache, interconnect, main memory) and mitigation approach (scheduling-based, regulation-based, hardware-based). This contextualizes the contributions of the present work within the broader landscape of memory interference research.

The first research track, presented in Chapter 4, focuses on detailed characterization of memory interference at multiple levels of the memory hierarchy. Rather than isolating single causes, this chapter systematically decomposes the overall interference into contributions from main memory bandwidth saturation, cache thrashing, and micro-architectural phenomena. Through evaluation on representative NVIDIA and Xilinx platforms, the chapter challenges the conventional assumption that synthetic read-intensive workloads represent worst-case interference, demonstrating that platform-specific characteristics fundamentally shape interference effects.

The second research track, presented in Chapter 5, addresses the previously understudied reconfiguration component of Memory Bandwidth Management Schemes. This chapter presents the first comparative analysis of synchronous and asynchronous bandwidth reconfiguration techniques, evaluating their effectiveness on dynamically changing mixed-criticality workloads. The analysis demonstrates that asynchronous, event-driven techniques achieve superior control granularity, enabling microsecond-scale adaptation in systems where task sets change frequently.

The third research track, presented in Chapter 6, extends memory interference mitigation to heterogeneous accelerators, proposing a novel spatial-domain approach for GPU bandwidth management. Operating at the Streaming Multiprocessor granularity and leveraging CUDA Green Contexts, this MBMS provides practical, portable GPU resource control. The chapter also

uncovers platform-specific pathological cases on modern NVIDIA hardware, revealing fundamental limitations of purely spatial-domain approaches and motivating hybrid mitigation strategies.

Finally, Chapter 7 synthesizes the findings across all three research tracks, discusses key insights regarding platform-specific interference and the importance of multi-level analysis, acknowledges limitations of the current work, and outlines three focused directions for future research tied directly to each of the main chapters.

Chapter 2

Background: COTS Heterogeneous Systems on Chip

As previously mentioned in Section 1, this thesis focuses on COTS HeSoCs, parallel hardware platforms capable of accommodating the demanding requirements of modern CPSs. In this Section, we describe the architectural model underlying contemporary HeSoCs, with particular focus on the memory hierarchy components and the hardware accelerators. Moreover, we present examples of relevant HeSoCs employed to implement CPSs.

2.1 Towards heterogeneous systems

Early computing systems operated as single-core processors with limited computational capacity: for decades, performance improvements relied on increasing clock frequencies. By the early 2000s, physical constraints – particularly power consumption and heat dissipation – made further frequency scaling impossible [91]. This prompted a fundamental shift toward the use of multi-core processors, where multiple independent cores execute instructions in parallel. However, also this trend reached a limit due to the number and power efficiency of the transistors installed on silicon: many modern

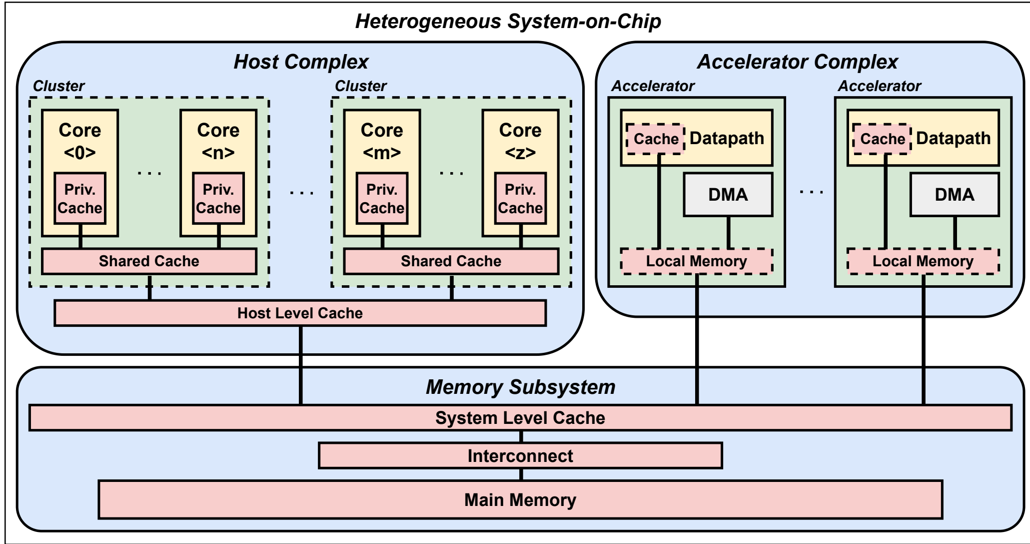


Figure 2.1: HeSoC Architecture Model.

computer chips have so many transistors that they cannot all be powered on simultaneously due to power consumption and thermal constraints. This limit, known as the *utilization wall* [111], motivated the integration of specialized hardware accelerators optimized for specific workload classes, which can achieve superior energy efficiency compared to general-purpose cores. ARM's *big.LITTLE* architecture, introduced in 2011, first exemplified this paradigm by pairing energy-efficient low-power cores with high-performance cores to optimize both throughput and power consumption [67]. Today, heterogeneity represents the industry standard for embedded platforms, enabling high-performance at limited thermal and power envelopes.

2.2 Architectural Model

Modern COTS HeSoCs generally adhere to the architectural model presented in Figure 2.1. The figure shows the main components of a modern HeSoC: the host complex, the accelerator complex, and the memory subsystem.

2.2.1 Host complex

The host complex represents the primary control entity of an HeSoC, which executes the main application logic, coordinates accelerator tasks, and manages the system resources. In general, it comprises one or more *Central Processing Unit* (CPU) cores, possibly organized in clusters.

CPU cores are the basic operational units of the system. In modern HeSoCs, they implement various *Instruction Set Architectures* (ISAs), including both *Reduced Instruction Set Architecture* (RISC) and *Complex Instruction Set Architecture* (CISC) designs. Note that different cores within the host complex can implement distinct ISAs: high-performance cores maximize computational throughput, while energy-efficient cores prioritize power consumption. This architectural heterogeneity enables effective performance-power trade-offs across the system.

Each CPU core integrates one or more levels of private cache hierarchy. These private caches store frequently accessed data and instructions, reducing access latency. Complementing these per-core caches, all CPU cores share a common Host Level Cache. Moreover, they also share a System Level Cache with the accelerator complex. These caches enable efficient data sharing among cores without requiring access to main memory.

2.2.2 Accelerator complex

The accelerator complex comprises specialized hardware units (accelerators) integrated into the SoC to execute compute-intensive workloads with superior performance and energy efficiency compared to the host complex. Unlike general-purpose processors, accelerators employ domain-specific architectural optimizations to solve particular classes of problems. Common accelerator types found in modern HeSoCs include Graphics Processing Units (GPUs) for massively data-parallel workloads, Digital Signal Processors (DSPs) for signal and image processing, or Field Programmable Gate Arrays (FPGAs) for customizable hardware acceleration.

Architecturally, each accelerator comprises two core components: the accelerator *datapath*, which executes compute operations, and the *Direct Mem-*

ory Access (DMA) engine, which manages data movement. The accelerator *datapath* and the DMA can execute concurrently, enabling simultaneous computation and data movements. Optionally, accelerators can include local caches and scratchpad memories – orders of magnitude faster than the main memory – that provide fast access to frequently- and recently-accessed data.

2.2.3 Memory subsystem

The memory subsystem of an HeSoC stores volatile data necessary for computation, enabling efficient communication across the various compute units within the platform. It is organized as a deep, shared hierarchy where the upper levels – closer to the compute units – offer the lowest latency, while the lower levels provide larger storage but with higher access latency. The main components of this subsystem include caches (which may be private to individual cores or shared among multiple units), the interconnect fabric that routes data, and the main memory, which is managed by the memory controller.

Caches

Caches are small, high-speed memory components located close to the compute units, designed to store recently and frequently accessed data for rapid retrieval. Their operation is based on the assumption that memory locations accessed in the past are likely to be accessed again in the next future (*temporal data access locality*), along with adjacent locations (*spatial data access locality*). Leveraging these locality principles, caches reduce the frequency of expensive accesses to slower main memory, significantly enhancing overall system performance.

System Interconnects

System interconnects – simply labeled as *Interconnects* in Figure 2.1 – serve as the primary communication backbone linking compute units and peripheral controllers to the main memory. These components receive, multiplex,

and forward memory requests from the various initiators to the main memory. In particular, compute units and peripherals act as *masters* (generating transaction requests), while the main memory modules function as *slaves* (servicing these requests).

Historically, on-chip communication relied on shared busses or simple point-to-point connections. However, as the number of integrated compute units increases, these traditional topologies hit severe performance and scalability bottlenecks [77]. To overcome these limitations, modern HeSoCs predominantly employ *Network-on-Chip* (NoCs) [26] or *crossbar-switch* architectures.

In NoC-based designs, communication nodes (i.e., *masters* and *slaves*) leverage network interfaces to send and receive network packets using standard protocols (e.g., AMBA AXI or CHI [10]). These packets traverse a distributed fabric of routers and links to reach their destination. Conversely, crossbar architectures utilize a matrix-based topology to establish direct, non-blocking connections between any possible initiator and any possible target. Despite these structural differences, both architectures enable concurrent data transfers, thereby delivering the high bandwidth and scalability required by contemporary HeSoCs.

Main Memory

The main memory acts as the primary volatile storage for all heterogeneous compute units within the HeSoC, providing the working space for active applications. This memory is typically implemented using Dynamic Random Access Memory (DRAM) technology. Internally, the DRAM memory is structured in independent data *banks*, the actual memory modules that store data. This multi-bank organization is critical for performance, as it allows for *Bank-Level Parallelism*, enabling multiple memory operations to proceed simultaneously across different banks.

Access to main memory is managed by a Memory Controller (MC). This component acts as the interface between the system interconnect (e.g., NoC or crossbar) and the physical memory. Its primary function is to schedule and

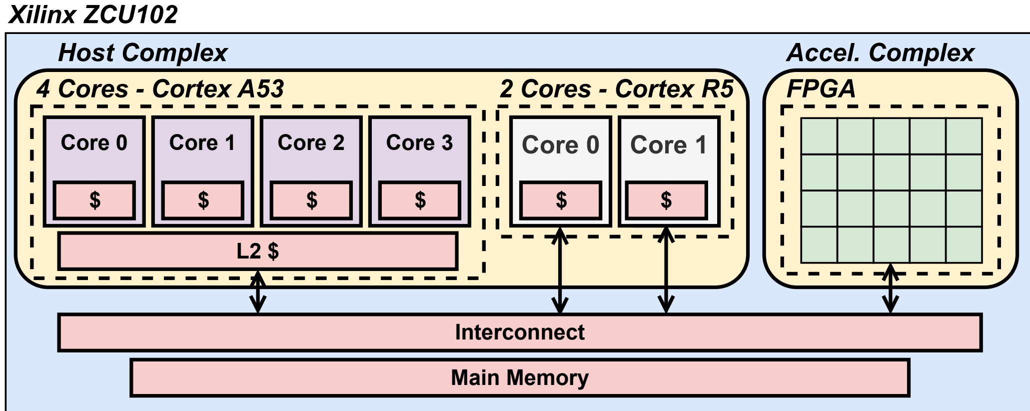


Figure 2.2: Architecture of Xilinx ZCU102 development platform.

arbitrate incoming memory requests from various initiators – such as CPUs, GPUs, and DMA engines – to optimize bandwidth and minimize latency. By leveraging the DRAM's internal banking structure, the controller can issue multiple memory requests in parallel to different banks, effectively masking latency of the physical memory to the initiators.

2.3 Relevant HeSoCs

As representative examples of the architecture described in Section 2.2, we hereby present HeSoCs from NVIDIA and Xilinx, the two leading vendors in this market segment. Specifically, we describe two Xilinx platforms : the Xilinx UltraScale+, and the Xilinx Versal ACAP; and four platforms by NVIDIA: the NVIDIA Jetson TX2, the NVIDIA Jetson Xavier, the NVIDIA Jetson AGX Orin, and the latest commercially-available platform from NVIDIA, the NVIDIA Jetson AGX Thor. In the following sections, we present the architectural detail of the various platforms.

2.3.1 Xilinx UltraScale+

The UltraScale+ is a family of heterogeneous SoCs produced by Xilinx. Among the various platforms in this family, the ZCU102[1] development board

[1] https://docs.amd.com/v/u/en-US/ug1182-zcu102-eval-bd

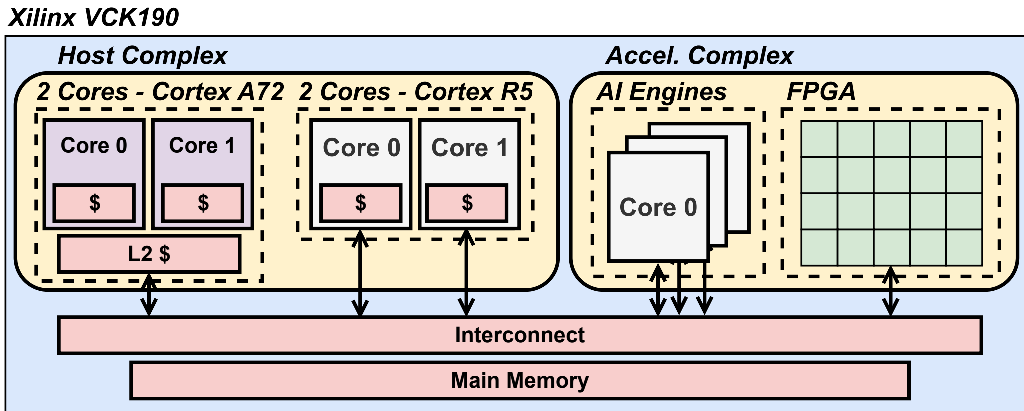


Figure 2.3: Architecture of Xilinx VCK190 development platform.

is one of the most widely adopted. This work specifically evaluates the Xilinx ZCU102.

The architecture of the ZCU102 platform is shown in Figure 2.2. Within the host complex, this platform integrates two processing islands: a quad-core ARM Cortex-A53 general-purpose CPU, running at 1.2 GHz; and a dual-core ARM Cortex-R5 real-time processing unit. The real-time processing unit can execute both as a dual-core cpu or in *lockstep*, to implement redundancy. The cortex A53 CPU implements 64KB L1 cache per core (32KB for instruction and 32KB for data) and a 1 MB shared L2 cache, both configured with a 64-byte cache line size.

Within the accelerator complex, the ZCU102 provides a FPGA, a reconfigurable accelerator allowing system-designers to implement custom and application-specific accelerator architectures.

The memory subsystem consists of 4 GB LPDDR4 memory with a 128-bit bus width operating at 1.3 GHz.

2.3.2 Xilinx Versal ACAP

As the UltraScale+, Versal ACAP is a family of HeSoCs produced by Xilinx. Specifically, this work evaluates the VCK190[2] development platform, a

[2]https://docs.amd.com/r/en-US/ug1442-vck190-base-trd/Versal-ACAP-Device-Architecture

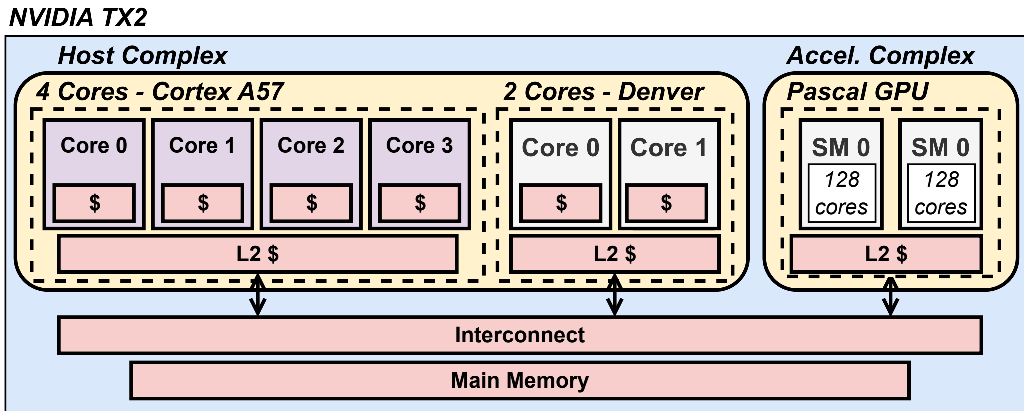


Figure 2.4: Architecture of NVIDIA TX2.

common platform belonging to the Versal ACAP family.

Figure 2.3 shows the architecture of the VCK190 platform. Within the host complex, the VCK190 combines a dual-core ARM Cortex-A72 CPU, operating at 1.35 GHz, and a dual-core Cortex-R5 real-time processing unit. The A72 cores feature 48 KB instruction and 32 KB data L1 caches per core, complemented by 1 MB shared L2 cache. All cache memory components are configured with a 64-byte cache line size. The accelerator complex is expanded with respect to the ZCU102 platform: the VCK190 provides both a FPGA and several *Intelligent Engines*, special cores (*very long instruction word* cores) aimed at improving the performance of Deep Learning workloads.

Memory is supplied as 8 GB DDR4, accessed via a NoC interconnect.

2.3.3 NVIDIA Jetson TX2

The Jetson TX2[3], from now on referred to as TX2, is an NVIDIA HeSoC released in 2017. Figure 2.4 illustrates its architecture.

Within the host complex, the TX2 employs a unique dual-island CPU architecture with asymmetric core specialization. The first island comprises four ARM Cortex-A57 cores clocked at 2 GHz, each with 48 KB instruction and 32 KB data L1 cache, and 2 MB shared L2 cache. The second island

[3]https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/

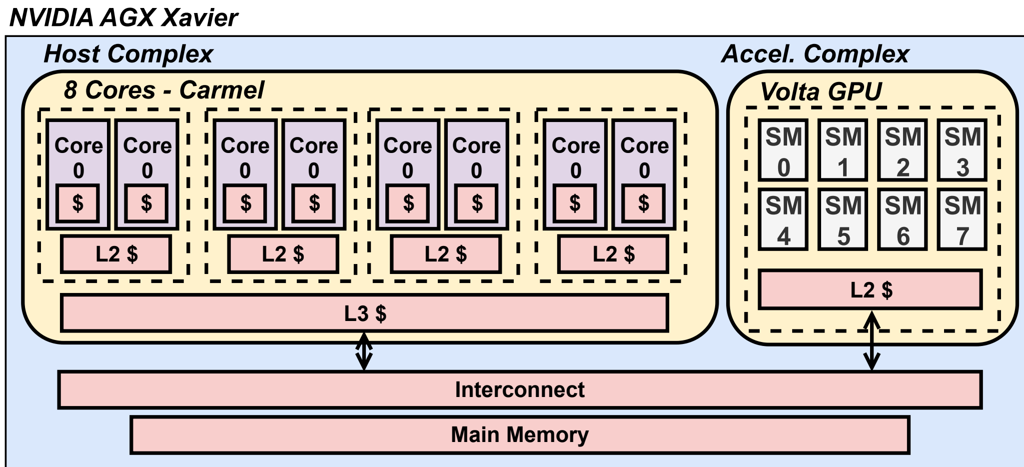


Figure 2.5: Architecture of NVIDIA AGX Xavier.

contains two *Denver* cores – an Arm-V8 compliant core directly designed by NVIDIA – also clocked at 2 GHz. The Denver CPU features more computational power compared to the Cortex-A57. Moreover, it is fitted with larger per-core caches: 128 KB instruction and 64 KB data L1 caches, and a 2 MB shared L2 cache. Both islands use 64-byte cache lines. In this case, NVIDIA implements the so called *big.SUPER* architecture, where the Denver cores are used to handle more demanding workloads.

Within the accelerator complex, the platform integrates a GPU based on the *Pascal* architecture. The GPU features 2 SMs, integrating 128 cores each. SMs are clocked at 1.3 GHz.

The memory subsystem features an 8 GB main memory with a 128-bit bus width operating at 1866 MHz maximum frequency.

2.3.4 NVIDIA Jetson Xavier

The Jetson AGX Xavier[4] represents the evolution of the Jetson TX2 platform. It was released by NVIDIA in 2018. Figure 2.5 illustrates its architecture.

Within the host complex, the AGX Xavier platform features 8 *Carmel*

[4]https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/

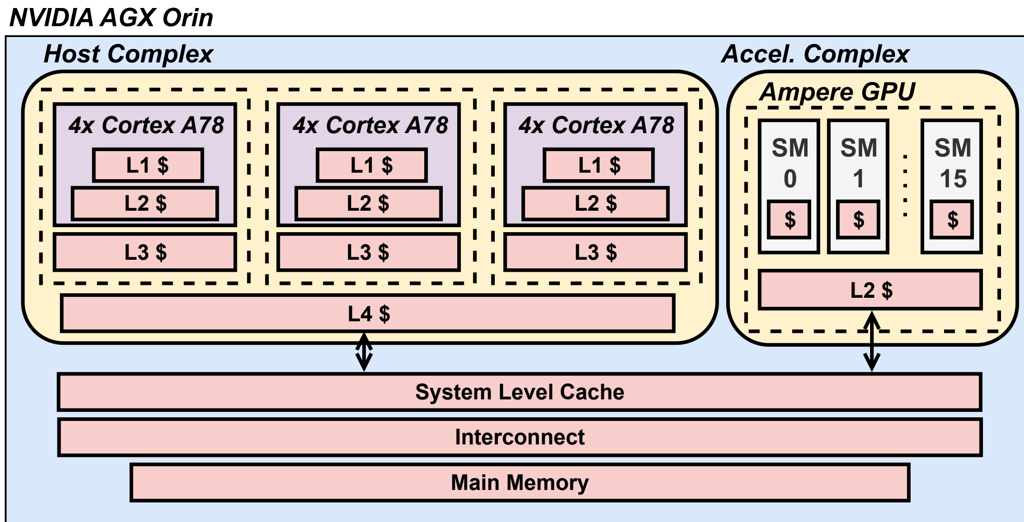


Figure 2.6: Architecture of NVIDIA AGX Orin.

cores organized into 4 clusters of 2 cores. The *Carmel* core is an ARM v8.2-compliant CPU designed by NVIDIA. On this platform, this CPU is clocked at 2.2 GHz. Each core implements 64 KB L1 cache, with 2 MB L2 cache shared per cluster and 4 MB L3 cache shared system-wide, maintaining 64-byte cache lines.

Within the accelerator complex, the platform integrates a GPU based on the *Volta* architecture. This GPU features 8 SMs, each featuring 64 cores clocked at 1.37 GHz.

The memory subsystem supplies 16 GB LPDDR4 main memory through a 256-bit bus clocked at 2133 MHz maximum frequency.

2.3.5 NVIDIA Jetson AGX Orin

The Jetson AGX Orin[5], evolution of the Xavier platform, was released by NVIDIA in 2022. Figure 2.6 shows its architecture.

Within the host complex, the Jetson AGX Orin comprises 12 ARM A78AE cores organized into 3 clusters of 4 cores, each running at 2.265 GHz. Each core maintains dedicated 64 KB instruction and 64 KB data L1

[5]https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/

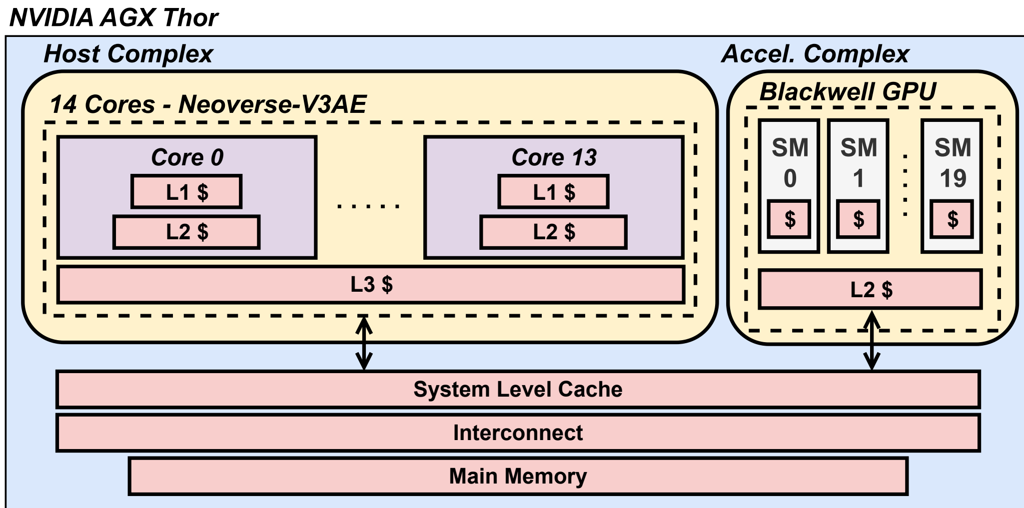


Figure 2.7: Architecture of NVIDIA AGX Thor.

caches along with 256 KB L2 cache. Additionally, each cluster shares 2 MB L3 cache, with 4 MB cache shared globally, all using 64-byte cache lines.

Within the accelerator complex, the Jetson AGX Orin offers a GPU based on the *Ampere* architecture. The GPU features 16 SMs, each equipped with 128 cores clocked at 1.3GHz. Each SM features 192KB of L1 cache, while all SMs share 2MB L2 Cache.

Memory is supplied as 64 GB LPDDR5 with a 256-bit bus width at 3.2 GHz maximum frequency.

2.3.6 NVIDIA Jetson AGX Thor

The Jetson AGX Thor[6] is the latest platform released by NVIDIA, made available in 2025. Figure 2.7 illustrates its architecture.

Within the host complex, the Jetson AGX Thor integrates 14 ARM Neoverse-V3AE cores operating in a single cluster at 2.6 GHz. Each core implements 64 KB instruction and 64 KB data L1 caches along with 2 MB L2 cache per core. Moreover, all the cores share a 16 MB shared L3 cache system-wide. All cache memory components are configured with 64-byte

[6] https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-thor/

cache line size.

Within the accelerator complex, the AGX Thor offers a GPU based on the *Blackwell* architecture. This GPU offers 20 SMs, each featuring 128 cores clocked at 1.57GHz. Each couple of SMs share up to 256KB of L1 cache. Moreover, the official reference manual also reports the presence of a shared L2 Cache. However, at the time of writing, no precise information about the size is found.

The memory subsystem provides 128 GB LPDDR5 through a 256-bit bus width at 4266 MHz maximum frequency.

Chapter 3

Related Work

Memory interference in multiprocessor systems has been studied since the 1970s [106], but has only emerged as a critical concern in recent times [85]. Over the last two decades, the scientific community has undertaken considerable work to analyze the problem of memory interference. Several works in the literature study how this problem arises in modern HeSoCs; other works focus on quantifying the negative effects that this problem can generate; and others provide mitigation strategies. In this section, we provide a comprehensive overview of these efforts, organizing the literature to mirror our own contributions: first, we begin by discussing works that characterize memory interference and quantify its effects on modern heterogeneous systems. Then, we focus on works that provide mitigation strategies, targeting memory interference that happens both within the *host* complex (i.e., among CPU cores), and between the *host* and *accelerator* complexes.

3.1 Memory Interference Characterization

In modern HeSoCs, interference can occur at all levels of the shared hierarchy: cache, system interconnect, and main memory [81, 45].

3.1.1 Shared Cache Level

At the shared cache level, memory interference occurs mainly because the limited space cannot accommodate the instructions [117, 61] and the data [86] needed by all compute units. When competing for cache space, the compute units *evict* data needed by other units, forcing them to access main memory and incurring additional latency. This problem has proven to be of critical importance, resulting in severe performance slowdowns for tasks contending for cache space [71]. For instance, Radojkovic et al. document execution time slowdowns reaching 14× on tasks co-scheduled on an Intel Atom processor [98]. Beyond contention for cache space itself, compute units also compete for other shared resources at the cache level: resources like *Miss Status Holding Registers* (MSHRs) and *Writeback Buffers* (WB), needed to implement non-blocking caches [73], are also contended [115]. Bechtel et al. report slowdown factors up to 10× and 345× when co-scheduled tasks contend MSHRs and WBs, respectively [22]. Interference at the shared cache level is not only due to contention: cache coherence protocols can also induce additional traffic and, in turn, interference [63]. Gracioli et al. study this problem, reporting that coherence-induced traffic can make parallel execution up to 4× slower than sequential execution [55].

Although cache sharing between the host and accelerator complex can provide benefits [54], it also introduces interference between these two major components [23]. Brilli et al. report slowdowns up to 3× when shared cache is contended by the CPU cores and the GPU on a NVIDIA TX2 platform [31]. Moreover, simulation-based studies by Wen and Zhang further reveal that cache coherence protocol overhead is a primary bottleneck in CPU-GPU systems, with the coherence directory controller becoming saturated during concurrent execution [116].

3.1.2 System interconnect and Main Memory Level

At the system interconnect and main memory level, memory interference arises because such components, which provide limited bandwidth, cannot serve requests coming from all compute units concurrently [12]. In particular,

the worst interference occurs reportedly at the main memory controller level [123, 62], where heavy optimizations (e.g., request reordering and batching) introduce latency, and requests pile up.

Most research works characterizing system interconnect and main memory interference focus on *host* complex interference – that is, interference generated solely by CPU core traffic. Two complementary approaches have emerged in this domain: empirical benchmarking and mathematical modeling. Empirical studies rely on extensive benchmarking to characterize system interconnect and main memory interference [46]. Bansal et al., for example, characterize the Xilinx ZU9EG platform using both *read* and *write* memory-intensive micro-benchmarks on CPU cores [20]. Building on the same platform, Stevanato et al. extend the approach by introducing an automated tool that systematically tests diverse memory access patterns to analyze interference effects at the main memory level [109]. In contrast, other works leverage mathematical modeling to derive provable bounds on memory interference-induced delays. Kim et al. develop an analytical model of main memory to bound worst-case delays [70], while Andreozzi et al. employ a Mixed Integer Linear Programming (MILP) formulation to compute exact worst-case delays [14].

In recent times, the increasing use of hardware accelerators motivated researchers to also study the interference generated by the hardware accelerators within the *accelerator* complex. Early work by Cavicchioli et al. analyzed memory contention across multiple NVIDIA Tegra platforms (K1, X1), demonstrating that GPU memory-intensive operations can increase CPU memory latency by up to 4×, while CPU activity degrades GPU performance up to 2× [39]. More recent studies have extended this analysis. Capodieci et al. tracked the evolution of memory interference across three NVIDIA Tegra generations (Nano, TX2, Xavier), showing that although newer platforms improve memory bandwidth, interference remains critical: combined CPU-accelerator contention causes latency increases up to 45× [36]. On FPGA-based platforms, Mattheeuws et al. characterized interference on Xilinx ZU9EG system and demonstrated CPU slowdowns reaching 19× when accelerators stress the DRAM [82].

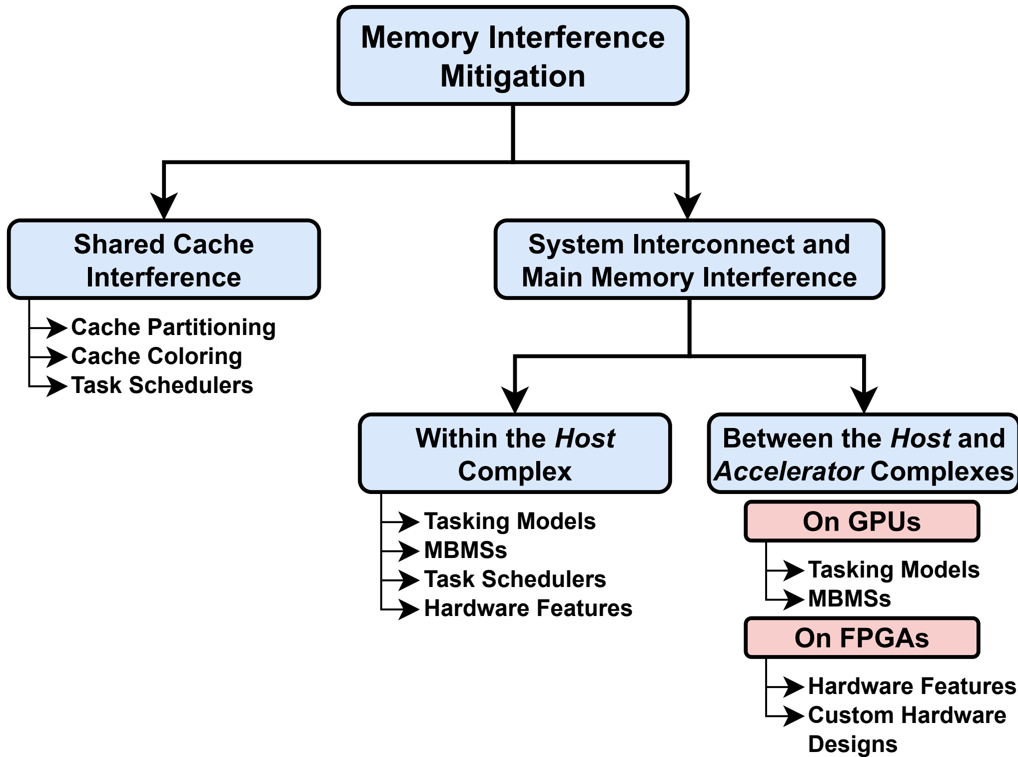


Figure 3.1: Categorization of memory interference mitigation techniques.

3.2 Memory Interference Mitigation

In the literature, research works providing memory interference characterization serve as a first and fundamental step not only in proving the feasibility of mixed-criticality CPSs [87, 13], but also in providing efficient mitigation techniques – the core of this work. Mitigation strategies depend on the type and source of memory interference to be mitigated. For this reason, and to improve the clarity of presentation, we organize this section into multiple subsections, each targeting a specific type of memory interference. A visual representation of the categories of approaches discussed in this section is provided in Figure 3.1.

3.2.1 Shared Cache Interference

Cache-level interference has motivated the development of multiple mitigation strategies. These approaches generally fall along a spectrum ranging from hardware- to software-based isolation strategies, with scheduling techniques complementing both.

At one end of this spectrum lie hardware-based cache partitioning approaches. With these approaches, cache *ways* or *sets* are physically partitioned among compute units, providing strict isolation guarantees. Intel *Cache Allocation Technology* (CAT) [96] and ARM *Memory Partitioning and Monitoring* (MPAM) [16] represent modern hardware support that allows fine-grained implementations of said partitioning approaches. However, static hardware-enforced cache partitions can reduce the possibility of compute units to share data at the cache level, in turn increasing communication latency. To overcome this limit, El-Sayed et al. present *KPart* [47], a hybrid approach that, relying on hardware-based cache partitioning, clusters application that share data onto the same cache partition, reducing interference while still maintaining data sharing capabilities.

For systems lacking dedicated hardware support, software-based cache *coloring* offers a practical alternative [126, 110]. The approaches based on cache-*coloring* allocate memory pages (addresses) that do not collide in cache to different applications. This reduces cache *evictions* and, in turn, ensures cache space isolation. Within this category of approaches, Ye et al. propose *COLORIS* [118], a cache-coloring mechanism that allows frequent reconfiguration of cache partitions based on tasks memory demands. These software techniques integrate seamlessly with standard operating system page allocators and require minimal hardware modifications.

Beyond hardware- or software-based cache partitioning, cache interference-aware scheduling can significantly reduce cache interference by controlling which applications execute concurrently [28, 59]. These approaches reduce *eviction* interference without explicit partitioning. Recent work by Yi et al. propose *CADE* [119], a scheduling technique that handles taskset expressing dependencies as Direct Acyclic Graphs (DAGs). This approach maps tasks

on cores with favorable cache state, improving the overall cache hit ratio and, in turn, the task performance.

3.2.2 System Interconnect and Main Memory Interference within the Host Complex

The research community has carried out significant work to mitigate system interconnect and main memory interference generated by the *host* complex [80]. In particular, researchers proposed a plethora of approaches that are particularly focused on controlling CPU core-level memory bandwidth consumption, in turn reducing the interference between cores. These approaches act at different abstraction levels. At the highest abstraction level, interference-aware tasking models and scheduling techniques try to mitigate interference by avoiding simultaneous execution of memory intensive tasks. At lower abstraction level, *Memory Bandwidth Management Schemes* enforce specific, per-core bandwidth regulation. Lastly, other approaches make use of QoS-enabling features provided directly by the hardware.

Tasking Models

At the highest abstraction level, the *Predictable Execution Model* (PREM) stands out as a fundamental tasking model to achieve memory access isolation in multicore CPUs [92, 93]. Within this model, tasks execution is divided into memory phases and compute phases, where memory prefetch and write-back operations are strictly separated from computation. By enforcing a platform-level schedule where only a single core is permitted to execute a memory phase at any given time, PREM ensures that memory accesses occur without interference from other cores, thereby achieving timing performance isolation. The PREM tasking model has been continuously studied over the years: Alhammad et al. extended its design to multithreaded applications [7], while Pagetti et al. more recently demonstrated how PREM-compliant tasks can be automatically generated through compiler transformations [89].

Memory Interference-Aware Schedulers

Beyond interference mitigation at the task model level, memory-aware task scheduling offers a complementary approach to minimize memory interference. Memory-aware schedulers leverage knowledge of task memory characteristics to determine optimal core-to-task mappings and execution sequences: these schedulers enforce the isolated execution of *critical* tasks that are sensitive to memory-interference, thereby minimizing the timing performance impact. A notable example of these techniques is presented by Aceituno et al. [1], which demonstrate that incorporating hardware resource contention into schedulability analysis enables previously unschedulable mixed-criticality tasksets to meet their *deadlines*. Complementary to this approach, Ali et al. propose the *RT-Gang* scheduler [9], a scheduling technique that groups tasks into *gangs* that do not interfere in memory, and schedules *gangs* one-at-a-time.

Memory Bandwidth Management Schemes

Memory Bandwidth Management Schemes (MBMSs) – central to this work – represent a common category of techniques for addressing memory interference. These approaches operate on a simple but effective principle: by *regulating* (limiting) memory access bandwidth for *best-effort* tasks, they ensure that *critical* tasks can access memory undisturbed. This, in turn, enables *critical* tasks to meet *deadlines* while still allowing *best-effort* tasks to make progress.

MBMSs can be broadly classified into two categories: software- and hardware-enabled. Software-enabled MBMSs make use of limited device-specific hardware features – like CPU cores *Performance Monitoring Units* (PMUs) – to enforce bandwidth *regulation*. A milestone in this category is represented by *MemGuard* [124, 125], a widely-studied MBMS that periodically *regulates* the bandwidth of CPU cores so that the memory interference in the system is mitigated. This MBMS is implemented and used both at Operating-System (OS) level [108, 4] and hypervisor level [84], and is followed by many other works in the literature. Yun et al. present BWLOCK

[121], MBMS that adopts *MemGuard*'s *regulation* technique but applies it selectively during *Memory-Critical Sections* (MCSs) of *critical* tasks. Saeed et al. propose a MBMS that observes memory controller utilization, and consequently enforces bandwidth *regulation* on *best-effort* tasks [101]. The same authors further present a complementary approach that continuously samples memory access latencies at the controller level, triggering bandwidth *regulation* on *best-effort* tasks when latencies become excessive [102]. In contrast, Flodin et al. adopt an event-driven strategy using *Inter-Processor Interrupts* (IPIs) triggered by *critical* tasks to dynamically activate bandwidth *regulation* on *best-effort* tasks [51].

On the other hand, hardware-enabled MBMSs make use of dedicated compute units to enforce bandwidth *regulation*. Within this category, recent work by Zuepke et al. presents *MemPol* [130], a MBMS that employs a low-power core or an FPGA to perform polling-based *regulation* and *reconfiguration* from outside the application cores. This approach cancels the overhead due to software-based *regulation* and drastically reduces the *reconfiguration* latency, demonstrating micro-second scale control granularity. Moreover, Izhbirdeev et al. propose *MemCoRe*, a MBMS that controls the memory bandwidth of the application cores using a cache-coherent FPGA [66]. In their article, they implement *regulation* using a token-based approach, while they *reconfigure* bandwidth based on the memory addresses accessed by the processing elements. In other words, they allocate a specific bandwidth per memory-address pool. This approach showed nanosecond-scale precision.

Solutions based on hardware QoS enabling features

The scientific community also tackled the memory interference problem at the hardware level, proposing predictable memory controllers [114, 6] and custom hardware components to enforce bandwidth *regulation* [50, 38, 56] or traffic shaping [127]. Other works utilize QoS-enabling hardware modules, which are platform-specific. Zini et al. analyze and employ ARM's specific features (i.e., ARM MPAM) to enforce memory bandwidth *regulation* [128]. Sohal et al. provide an analysis of the Intel's *Resource Director Technology* (RDT)

[107], while Farina et al. focus on Intel's *Memory Bandwidth Allocation* (MBA) [49] – one of the features included in Intel RDT.

3.2.3 System Interconnect and Main Memory interference between the ***Host-Accelerator*** Complexes

The memory interference mitigation strategies that involve the *accelerator* complex strongly depend on the type of hardware accelerator generating (or suffering) the interference. In the next sections we provide insights on mitigation strategies targeting platforms that rely on GPUs and FPGAs as main hardware accelerators.

Host Complex to GPU interference

Host-to-GPU system interconnect and main memory interference (and viceversa) are mitigated using approaches that are often related to the ones used for *host*-level interference.

Early work by Capodieci et al. proposes *SiGamma* [37], a MBMS protecting the CPU cores from GPU-induced interference. This approach assumes CPU tasks to comply with the PREM task model [92], idling GPU operations during CPU memory phases to prevent competing memory traffic. Extending this work, Forsberg et al. propose *HePREM* [52], an extension of the original PREM proposal in the context of heterogeneous SoCs. This approach proposes a compiler-based technique to make GPU code PREM-compliant. By scheduling memory phases of both CPU and the GPU tasks one-at-a-time, this approach ensures system-wide memory interference mitigation.

Another work by Ali et al. proposes *BWLOCK++* [8], an extension of the *BWLOCK* framework [121] to the GPU-based platform. BWLOCK++ essentially activates CPU-cores bandwidth *regulation* when a *critical* task executes a GPU task. This way, the *critical* task executing on the GPU accesses memory undisturbed. Moreover, Aghilinasab et al. extend *BWLOCK++* [2], configuring BWLOCK to dynamically change at runtime the memory bandwidth allocations for *best-effort* tasks based on the GPU advancement.

Using a radically different approach, Park et al. examine the feasibility of GPU-accelerated mixed-criticality automotive applications [90]. Their strategy mitigates inter-task interference by periodically accounting for memory accesses performed by each task – which utilize both CPU and GPU – and adapting bandwidth *regulation* in response.

Lastly, a number of works leverage hardware-specific features to enforce GPU-bandwidth *regulation*. Seals et al. propose *BandWatch* [104], a MBMS that provides system-wide bandwidth regulation for GPU-based HeSoCs. *BandWatch* is based on *MemGuard* to *regulate* the memory bandwidth of the CPU cores. Moreover, it leverages a hardware throttling feature available on the NVIDIA TX1 platform to *regulate* the GPU bandwidth. Specifically, the NVIDIA TX1 memory controller exposes a configurable parameter that enables fine-grained GPU bandwidth *regulation* across 32 discrete levels, allowing precise control of GPU memory consumption. This same approach is also used by Yun et al. to propose the *RT-Gang++* scheduler [120, 24], an extension of the *RT-Gang* CPU task scheduler [9] that handles tasks based on GPU-acceleration.

Host Complex to FPGA interference

Host-to-FPGA system interconnect and main memory interference are typically mitigated using approaches that either rely on hardware-specific *Quality of Service* (QoS) features or custom hardware designs deployed onto the FPGA.

Several works in the literature implement FPGA bandwidth *regulation* using hardware-specific features at different levels of the shared hierarchy. Garcia et al. leverage the QoS mechanism featured by the main memory controller of the ZU9EG platform [53] to prioritize the CPU memory traffic. With this setup they are able to ensure max 6% slowdown for the CPU tasks. Brilli et al. evaluate a system interconnect level feature – the ARM QoS-400 module [15] – to *regulate* the memory bandwidth of the FPGA on the ZU9EG platform [34]. The same approach is also used by Zini et al. to *regulate* the memory bandwidth consumed by other I/O peripherals [129]. However,

adopting QoS-enabling features to address this problem may be limited by their non-universal availability [105]. Each vendor implements proprietary solutions, and some platforms lack them altogether. Additionally, configuration latency can arise from the loose coupling between these components and the CPU. As a result, their use in closed-loop control schemes may constrain the achievable control frequency.

For this reason, researches propose several FPGA-deployable hardware designs to mitigate memory interference. Brilli et al. propose the use of *Controlled Memory Requests Injection* (CMRIs) to limit the FPGA-induced interference on the CPU tasks [32]. This approach – also used in other works to limit memory bandwidth GPU-based HeSoCs [40, 33] – enables the FPGA to exploit more than 40% of the available bandwidth, while keeping the timing impact (slowdown) on the CPU tasks below 10%. Moreover, another work by Brilli et al. proposes the *Runtime Bandwidth Regulator* [30], an FPGA-deployable soft-core that combines tight memory bandwidth monitoring [113] and *regulation*. Pagani et al. propose the *AXI Budgeting Unit* [88], an hardware design that implements periodic bandwidth *regulation* of the FPGA. Extending this work, Restuccia et al. present the *AXI interconnect* [99], providing a full system interconnect implementation that features both bandwidth *regulation* and improved fairness of the AXI interconnect [100].

Chapter 4

Taking a Closer Look at Host-Level Memory Interference Effects

As detailed in Section 2.2, vendors utilize varying design templates for HeSoCs, resulting in significantly different architectural couplings between compute units and the memory subsystem. This architectural diversity implies that memory interference arises differently for each platform, and cannot be captured by a single, platform-agnostic model. For any mitigation technique to be effective, it is important to rely on a deep understanding of architectural phenomena that lead to memory interference. Therefore, holistic, per-platform characterizations represent the first and fundamental prerequisite towards the design of effective and robust mitigation strategies.

The scientific literature presents extensive characterizations of hardware platforms [98, 29, 87, 22, 115, 72, 70]. However, many of these works suffer from a significant limitation: they often assume the worst case interference to be the one generated by synthetic read-intensive workloads that target the main memory (i.e., that completely bypass the cache hierarchy) [83, 5, 98, 70, 39, 36]. The intuition is that when using this type of workload for both the *core under test* (i.e. the core executing the workload that is subject to the slowdown being measured) and the cores that are generat-

ing interference, the system is subject to maximum interference (worst-case scenario). However, the massive degree of parallelism and architectural optimizations present in modern multicore CPUs may jeopardize this simplistic assumption, leading to inaccurate timing analysis and, in turn, incorrect memory interference mitigation techniques. This motivated a more in-depth study of the memory interference effects that parallel tasks are subject to when running simultaneously on the CPU cores of modern HeSoCs.

In this chapter, we provide a detailed characterization of the memory interference effects generated at various levels of the shared memory hierarchy by different synthetic and real-world workloads [57] co-scheduled on CPU cores. The evaluation, performed on two representative hardware platforms: *Nvidia TX2* and *Xilinx ZU9EG*, indicates that there is no universal worst-case memory access pattern, as the effects induced and suffered slowdowns strongly vary with the hardware. Moreover, the evaluation shows that memory interference happens differently at each level of the shared memory hierarchy, including main memory-interference effects, *cache trashing*, and all the way down to micro-architectural phenomena.

4.1 Background

In this section, we first recall the architectural templates characterizing the platforms targeted by this analysis. Subsequently, we examine the cache memory architectures of these platforms in detail, as they are one of the primary targets of our stress tests. Finally, we describe the set of workloads selected to drive the evaluation.

4.1.1 Architectural Model

Figures 4.1 and 4.2 show the architectural model of the HeSoCs we use in this analysis: a *Nvidia TX2* and a *Xilinx ZU9EG* platform. The figures are directly derived from Figures 2.4 and 2.2. However, in both figures, the *Accelerator* complex and the Host-level cores exceeding the main *Application Processing Unit* are *grayed out*, as we do not initially consider those compute

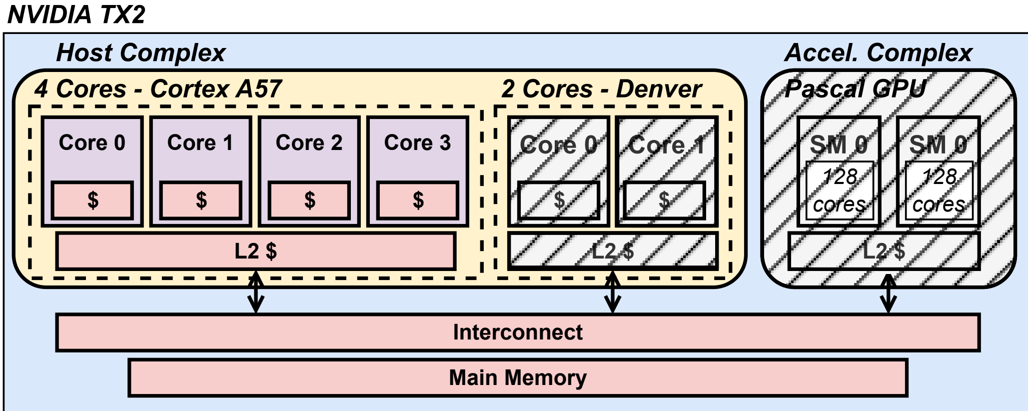


Figure 4.1: *Nvidia TX2* architectural model - *Host*-complex only.

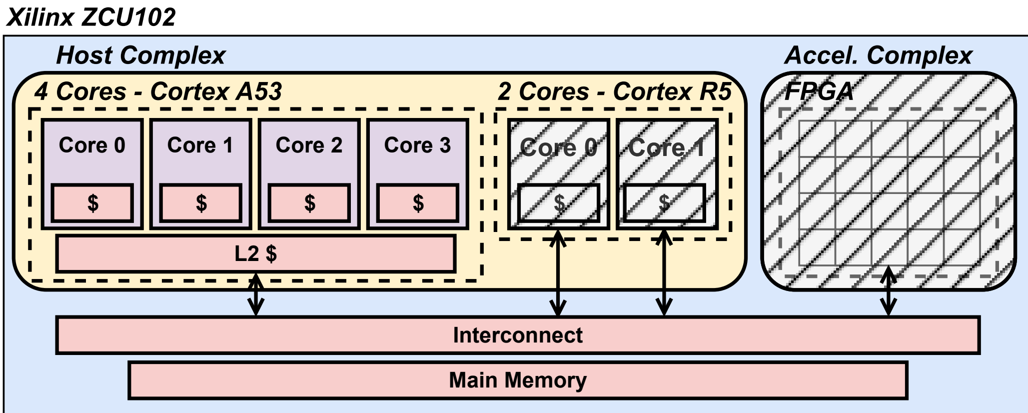


Figure 4.2: *Xilinx ZU9EG* architectural model - *Host*-complex only.

units. In this model, memory interference occurs at multiple levels: host-level shared cache, system interconnect, and main memory level. In this work, we focus at the host-level. Therefore, the shared host-level cache represents the cache level where interference (i.e., cache space contention) occurs. We refer to this cache as Last-Level Cache (LLC). For simplicity, we refer to cache-level interference (i.e., cache space contention) as *cache thrashing*, while we refer to system interconnect and main memory level as *main-memory interference*. For better clarity, we also report the hardware configurations of such platforms in Table 4.1.

	Xilinx ZU9EG	*Nvidia TX2*
CPU cores per cluster	4	4
Number of clusters	1	1
LLC size	1 MB	2 MB
LLC line size	64 B	64 B
Main Memory (DRAM) size	4 GB	8GB

Table 4.1: Relevant hardware information for our reference platforms.

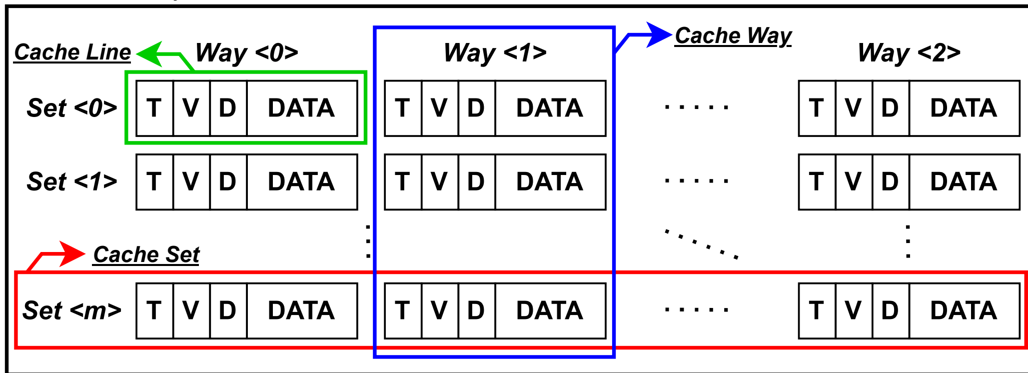


Figure 4.3: Cache Memory Architecture.

4.1.2 Cache Memory

In this analysis, we stress the cache memory components of the platforms we evaluate. For this reason, we hereby describe the structure and operation of such component.

Cache Architecture

Figure 4.3 shows the internal structure of an *n-way set associative* cache memory, the cache architecture used in both our reference platforms. An *n-way set associative* cache is internally organized as a large matrix of cache *lines*, i.e., fixed-size blocks of memory that store data. Cache *lines* are the basic units of storage and transfer between the cache and main memory. In an *n-way set associative* cache, *lines* are organized in *sets* (the horizontal rows) and *ways* (the vertical columns). Internally, each *line* is composed of a memory block (Figure 4.3, DATA), which stores the actual data, and several

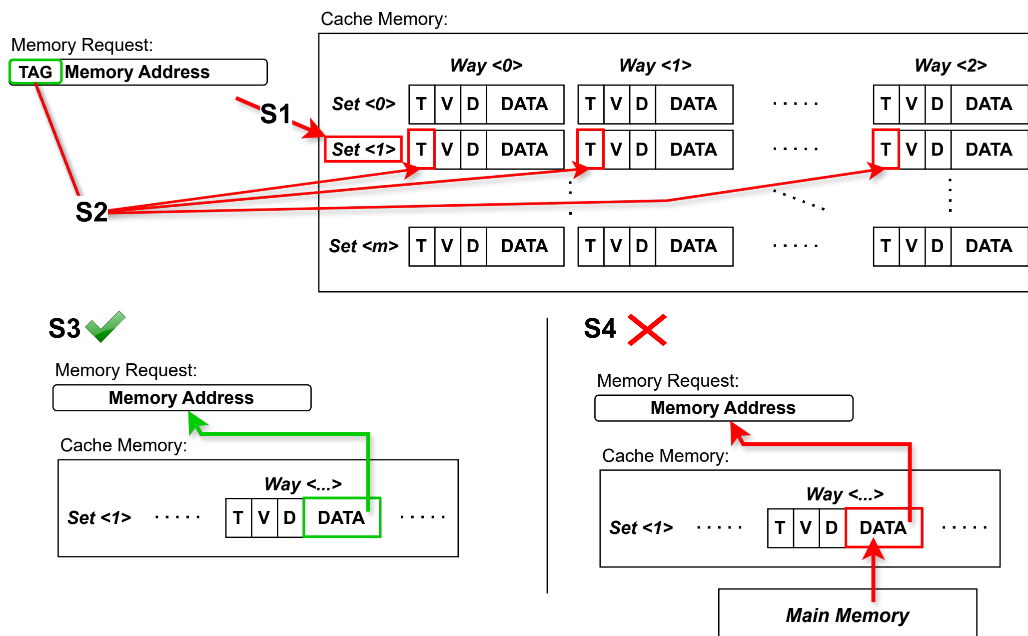


Figure 4.4: Cache Operation Diagram.

metadata used for cache maintenance operations. Among these metadata, there is the *tag* (Figure 4.3, T)), which contains the most significant bits of the memory address stored in the *line*. This metadata field enables the cache controller to precisely identify which data segment is stored in the cache *line*. Moreover, the metadata typically contains *control* bits: a *valid bit* (Figure 4.3, V) to indicate if the *line* contains initialized data; and a *dirty bit* (Figure 4.3, D) to track if the local copy of the data (i.e., the one residing in the cache *line*) has been updated relative to the copy stored in main memory.

Cache Operation

Figure 4.4 sketches the operation of an *n-way set associative* cache. At runtime, the cache memory receives *read* and *write* requests from the compute units. Upon receiving a request, the cache memory performs a *lookup*: it checks whether requested data (i.e.,. memory address) already reside in the cache. The *lookup* process is made up of two steps: first, the cache controller computes the target *set* for the requested memory address (Figure 4.4, step

S1). Note that each memory address is assigned by the cache controller to a single cache *set*. Then, the cache controller checks the requested memory address against the *tag* metadata of all cache *ways* in the corresponding *set* (Figure 4.4, step S2). If one of the *tag* matches the requested memory address, then the *line* holding that *tag* stores the requested data. Once the lookup is performed, if the requested data reside in the cache, then a *cache hit* occurs: the compute unit directly reads or writes the data in the resulting cache *line* (Figure 4.4, step S3). On the other hand, if the requested data do not reside in the cache, then a cache *miss* occurs and the data must be fetched from a lower-level memory: a lower-level cache or main memory. Upon a cache *miss*, the cache *evicts* (empties) a *line* in the *set* corresponding to the requested memory address. Then, it *refills* the requested data from a lower-level memory, places data into the previously *evicted line*, and serves the memory request (Figure 4.4, step S4). If the cache *line evicted* in this way contained updated data relative to the copy stored in lower-level memory (i.e., *dirty bit* set to 1) then the data is *written-back* to lower-level memory to maintain consistency. Otherwise, the data *evicted* from the cache *line* is simply discarded.

4.1.3 Workloads

In this analysis, we leverage both synthetic workloads and real-world benchmarks from the *PolyBench-ACC* benchmark suite [57].

Synthetic Benchmarking

Synthetic memory-access generators are commonly used to produce controlled interference and measure resulting slowdowns [33, 98, 40, 36, 87, 70]. They allow a wide spectrum of access patterns to be easily generated. In general, a synthetic benchmark [5, 83] 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*).

2. **Number of memory and CPU operations** ($mops, cops$). The synthetic benchmark issues *mops* memory operations of the specified type t in a loop. Between one memory operation and the next, *cops* compute operations (or simple no-ops) are executed. This allows to control memory-to-compute ratio and, in turn, to generate more or less interfering (or sensitive-to-interference) workloads.

3. **Address stride** (st). The distance between two consecutive memory accesses (i.e., the *stride*). It heavily influences the cache miss rate (i.e., the sensitivity to interference), as certain specific *stride* configurations may be chosen to avoid the effects of *line prefetching*, depending on what the benchmark wants to assess;

4. **Access pattern** (p). Most real-life workloads tend to generate *sequential* access patterns – which consist of constant, small *strides* [20, 40], which the main memory 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 main memory controller takes longer to handle. Note that very long constant strides behave as random patterns [20, 40];

5. **Memory footprint** (fp), the *footprint* of the data structure accessed by the benchmark heavily influences cache behavior, as data that fits entirely in cache is bound to generate a high number of *hits*, as opposed to data that exceeds the size of the cache.

Algorithm 1 describes the synthetic benchmark we designed. Reflecting the aforementioned characteristics, the algorithm takes as inputs:

1. The base address *addr* of the data structure on which the memory operations are performed.

2. The type t and number *mops* of memory operations to be performed on the data structure.

3. The number *cops* of compute operations to be done after every memory operation.

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

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

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

```
1  Function READ_MISS(src, reps)
       Input: src: read base address
2      for i ← reps do
3          SYNTH_BENCH(src, r, mops, cops, st, fp)
4      end
5  Function WRITE_MISS(dst, reps)
       Input: dst: write base address
6      for i ← reps do
7          SYNTH_BENCH(dst, ws, mops, cops, st, fp)
8      end
9  Function MEMSET(dst, reps)
       Input: dst: write base address
10     for i ← reps do
11         SYNTH_BENCH(dst, w, mops, cops, st, fp)
12     end
```

4. The stride *st* between the addresses of any two consecutive memory operations.

5. 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.

READ_MISS, ***WRITE_MISS***, and ***MEMSET*** benchmarks

For the analysis, we employ three specific types of traffic that can be generated using the synthetic benchmark:

- *READ_MISS*: traffic composed of only **load** operations that miss in LLC, triggering only cache refills.

- *WRITE_MISS*, traffic composed of only **store** operations that miss in LLC. In particular, these store operations are non-contiguous. This

traffic triggers both cache *refills* and *writebacks* (upon cache line eviction).

- *MEMSET*: traffic composed of contiguous **store** operations (i.e., stores that target contiguous addresses) that miss in the LLC. On our setup – the NVIDIA *Nvidia TX2* and the XILINX *Xilinx ZU9EG* – this traffic type enables the so called *read-allocate-mode* [18]. With this memory access mode, the CPU cores totally bypass the cache altogether, generating a traffic that targets exclusively the main memory. In particular, the full set of stores is posted into a *store buffer*, that is asynchronously copied to main memory, and the contents of the LLC remain unchanged. For the rest of this chapter, we refer to this type of traffic as *write-no-allocate* traffic (a name which better reflects the type of memory accesses generated).

These three specific types of traffic can be obtained by setting the parameters of the synthetic benchmark as follows: (i) the traffic type (t) is set to *read* for *READ_MISS* and *write* for *WRITE_MISS* and *MEMSET*; (ii) the memory footprint (*fp*) is set to be much larger than the LLC cache; (iii) the stride (*st*) is set to a multiple of the LLC line size for *READ_MISS* and *WRITE_MISS*, while it is set to 1 for *MEMSET*. These particular configurations generate the previously-described types of traffic. Algorithm 2 details the structure of *READ_MISS*, *WRITE_MISS* and *MEMSET*. In particular, the three instances of the synthetic benchmarks call the $SYNTH_BENCH$ function *reps* times inside a loop, with the aforementioned parameter configurations.

Many works in the literature [33, 98, 40, 36, 87, 70] have operated under the assumption that the worst-case access can be modeled by: (i) choosing $t = r$ or $t = ws$ (i.e., instantiating *READ_MISS* or *WRITE_MISS* traffic types); (ii) making the *stride* an integer multiple of the LLC line size with a *sequential access pattern*; (iii) choosing a *footprint* bigger than the full LLC size, as it removes the possibility of one of the cache levels intercepting the memory accesses with cache hits when the algorithm executes in a loop, thus causing a 100% miss rate (meaning DRAM operations). In this chapter, 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.

The Polybench-ACC benchmark suite

Synthetic benchmarks enable precise control of the generated memory traffic and, presumably, interference. However, the memory access patterns generated by synthetic benchmarks are limited to a subset of the ones generated by real-world workloads, and specific memory interference effects may not be triggered by them. For this reason, we also use real-world benchmarks in this chapter, the Polybench-ACC benchmark suite [57]. This benchmark suite is a collection of compute kernels that are commonly found inside larger programs belonging to different categories: data mining, stencils, and linear-algebra kernels and solvers. In our experimental setup we compile the suite for single-core execution with the default dataset size. For most benchmarks that corresponds to the STANDARD_DATASET size. For benchmark `correlation` the default setting corresponds to EXTRALARGE_DATASET. For `convolution-3d` and `convolution-2d` to LARGE_DATASET.

4.2 Evaluation

In this section, we detail the evaluation process and the results obtained by executing both the synthetic and real-world benchmarks on our setup.

4.2.1 Main Memory Bandwidth Saturation

We perform a preliminary experiment to study the phenomenon of *main memory bandwidth saturation*: the exact point at which the main memory is totally consumed, and cannot handle the requests of CPU cores with uniform latency anymore. To do so, on both our reference platforms, we execute

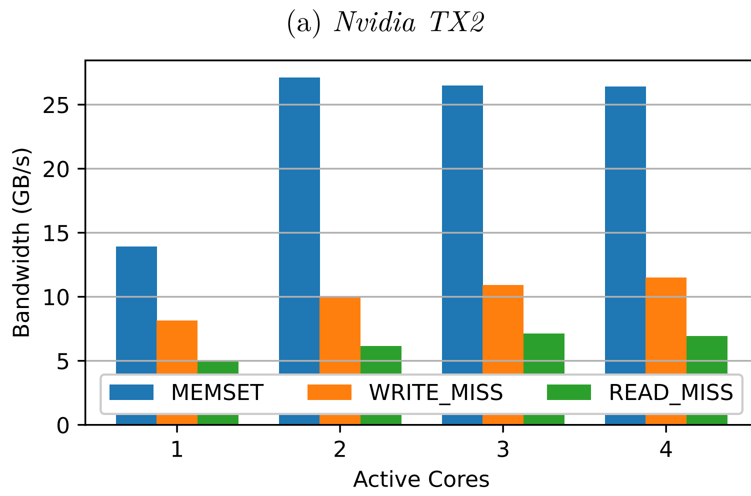
the *READ_MISS*, *WRITE_MISS*, and *MEMSET* traffics on an increasing number of cores, reporting the main memory bandwidth consumed.

Figure 4.5 shows the results of the experiment. In the plots, the X axis shows the number of cores employed, while the Y axis shows the cumulative bandwidth requested (average of 20 executions) for each of the three types of traffic. On both platforms, it is possible to observe *main memory bandwidth saturation* effects in the plots, as the requested bandwidth reaches a plateau for a number of active cores smaller than the total. Depending on the hardware platform, different types of traffic can reach saturation with different configurations. For instance, the *MEMSET* traffic reaches saturation when executed on a single CPU core on the *Xilinx ZU9EG*, whereas the other two types of traffic saturate as more CPU cores are involved. On the other hand, on the *Nvidia TX2*, the *MEMSET* traffic saturates memory bandwidth when executed on two CPU cores, and the *READ_MISS* traffic on three cores. In general, on both platforms, the three types of traffic consume very different bandwidth levels, and **the workload which generates the highest bandwidth is platform-dependent**.

4.2.2 Memory Interference Effects Analysis

In this section, we perform a second test, evaluating the effect of memory interference (i.e., the slowdown) on the tasks scheduled on the CPU. In particular, we measure the execution time of a *task under test* running on one of the cores, with the *interfering tasks* running on the remaining cores. The *task under test* is either one of our synthetic benchmarks or one of the Polybench-acc benchmarks (see Sec. 4.1), while the *interfering tasks* are one of our synthetic benchmarks. We include the Polybench-acc benchmarks as *task under test* in this interference analysis to better represent how real-world workloads may be subject to interference on our target platforms (*Xilinx ZU9EG* and *Nvidia TX2*). *Interfering tasks* access memory at a controllable intensity by changing the ratio between $Mops$ and *Cops* (see Sec. 4.1). By tuning this ratio, we regulate the percentage of total memory bandwidth that *interfering tasks* are allowed to consume. In the results, we report this percentage on

(a) *Nvidia TX2*



(b) *Xilinx ZU9EG*

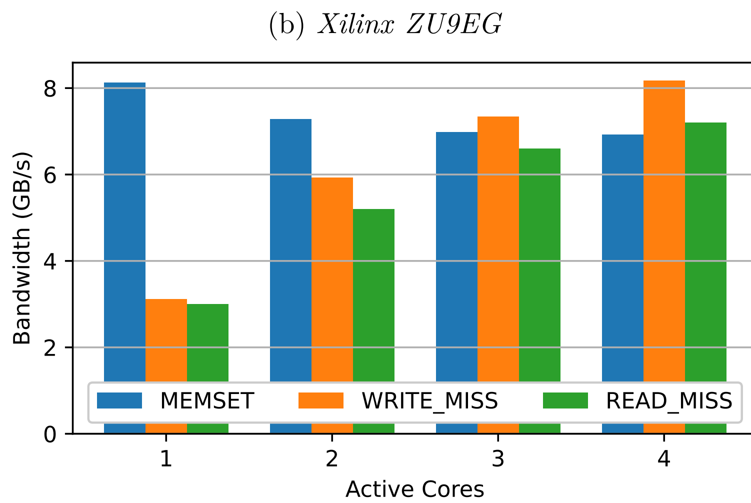


Figure 4.5: DRAM bandwidth reached by the three traffic types on our reference SoCs.

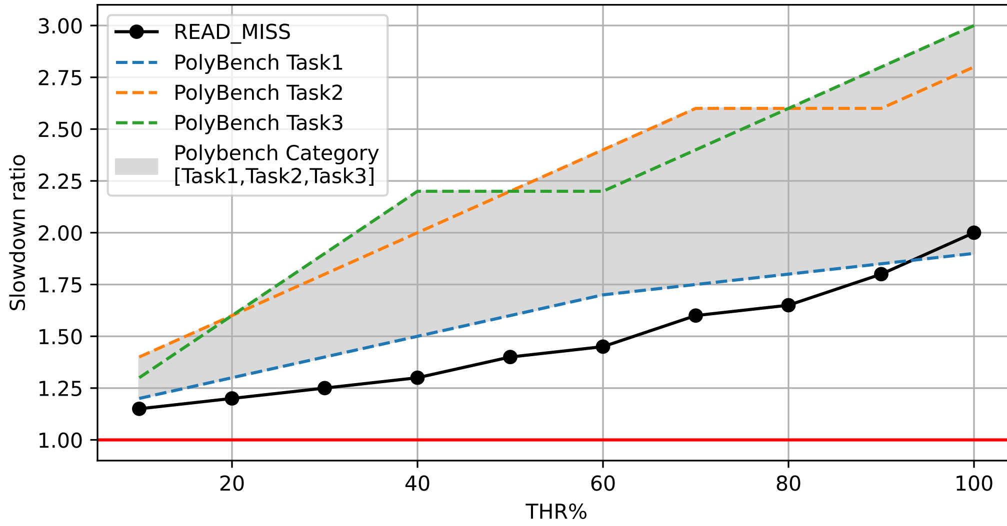


Figure 4.6: Example of slowdown plot including both the synthetic and PolyBench-acc benchmarks.

the X axis as throttling parameter *THR%*.

Evaluation Results Example

For better clarity, we present an example of how we report the slowdown results in Figure 4.6. In the plot, the slowdown of both the synthetic benchmarks and the PolyBench-acc benchmarks are represented as a function of the *THR%* parameter. Specifically, the slowdown suffered by the synthetic benchmarks running as *task under test* are shown as black curves. In contrast, the slowdown curve of the 30 Polybench-acc – also running as *task under test* – benchmarks are presented in the form of areas (i.e., grouped). Each area indicates a specific category of PolyBench-acc benchmarks, with a label indicating how many benchmarks fall into that specific category. For each category, the perimeter of the corresponding area (higher and lower) represents the highest and lowest values of all the slowdown curves of the Polybench benchmarks that belong to that category.

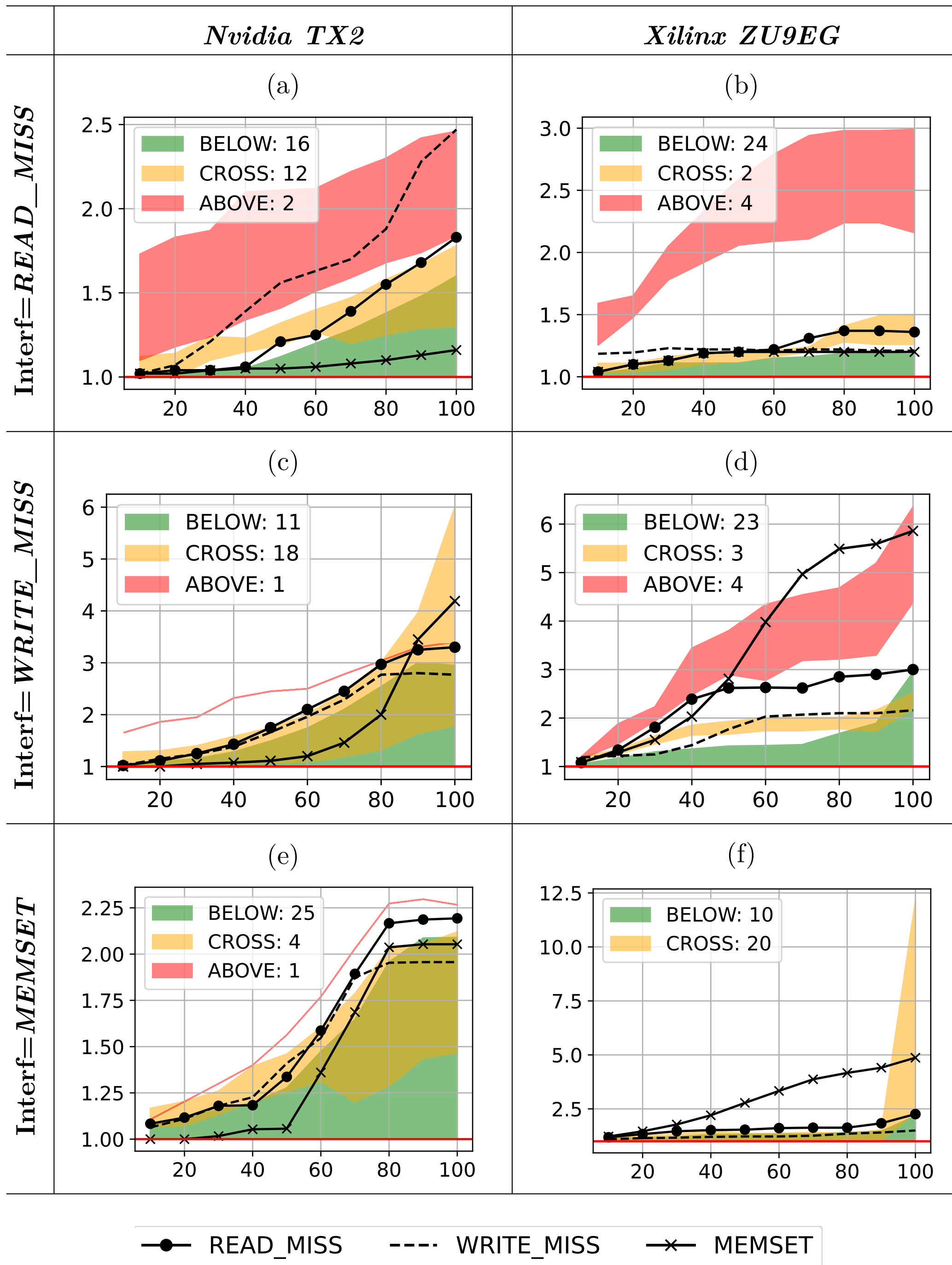


Figure 4.7: ***Nvidia TX2*** **(a,c,e)**, ***Xilinx ZU9EG*** **(b,d,f)**. Time increase for Synthetic and Polybench benchmarks. Note: each plot has a different maximum y axis value.

Evaluation Results

Figures 4.7a, 4.7c, 4.7e and 4.7b, 4.7d, 4.7f present the slowdown results due to the memory interference obtained on both the *Nvidia TX2* and the *Xilinx ZU9EG*, respectively. The plots are organized in a grid: one column for each platform, and one row for each interference traffic type. Each benchmark (synthetic benchmark or PolyBench-acc benchmark) is run 10 times due to the high amount of time some of the Polybench-acc benchmarks require to run, especially when subject to interference. The results we obtain are characterized by low standard deviation (all the results were within 5% of the average). Therefore, we report the average slowdown suffered by the benchmarks in the plots.

In each subplot, we divide the PolyBench-acc benchmarks into three categories based on their slowdown relative to the *READ_MISS* traffic:

1. **ABOVE**: benchmarks that suffer more slowdown than the *READ_MISS* traffic throughout the entire *THR%* range.

2. **CROSS**: benchmarks that suffer more slowdown than the *READ_MISS* traffic only for some *THR%* configurations.

3. **BELOW**: benchmarks that suffer less slowdown than the *READ_MISS* traffic throughout the entire *THR%* range.

The *READ_MISS* traffic (black curve with circular points) is used as the separation between categories as it is assumed by previous works to be the *task under test* most sensitive to memory interference [33, 98, 40, 36]. If that assumption were true, no benchmarks should fall into the **ABOVE** or **CROSS** categories in any plot of Figure 4.7. However, this is not the case.

Figure 4.7 shows that the *READ_MISS* traffic is never the *task under test* that is most interference-sensitive. In fact, in all subplots, at least one Polybench-acc benchmark is suffers more slowdown than the one suffered by the *READ_MISS* traffic. For instance, subplots 4.7a and 4.7b present the slowdown perceived by real and synthetic benchmarks when running in the presence of *READ_MISS interfering tasks*, an interference configuration

which some literature [98, 40, 36] considered to be the worst-case one. Subplot 4.7a 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 4.7b shows that, for the *Xilinx ZU9EG* platform, four Polybench fall within the **ABOVE** category, and two belong to the **CROSS** one. Note that these results are not captured by the assumption made in the previously cited works.
Subplots 4.7c and 4.7e show that the worst-case interference on *Nvidia TX2* is generated by *WRITE_MISS* (Subplot 4.7c), which causes R/W traffic due to the cache-line update operation. This result confirms previous literature [33, 87, 70]. However, subplots 4.7d and 4.7f show that on the *Xilinx ZU9EG* hardware the worst-case interference traffic type is *MEMSET* (Subplot 4.7f), 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 *WRITE_MISS* traffic.

Analysis Summary

The results presented by this test demonstrate that on the *Xilinx ZU9EG* and the *Nvidia TX2* platforms, *READ_MISS* is neither the workload causing the highest amount of interference (highlighted by subplots 4.7c and 4.7f) nor the one most sensitive to memory interference (as seen in all subplots of Figure 4.7). In fact, the results are highly platform-dependent, with the two analyzed HeSoCs having different worst-case *interfering tasks* (*WRITE_MISS* on the *Nvidia TX2* and *MEMSET* on the *Xilinx ZU9EG*). Figure 4.7, in general, highlights the need to do proper memory interference characterization, to determine the actual worst-case: it is not possible to make generalizations.

Finally, the results also show that some Polybench-acc benchmarks are more subject to interference than any of the synthetic benchmarks. In the next section, we explore why specific real-world benchmarks are more sensitive to memory interference than synthetic ones.

4.2.3 Last Level Cache thrashing

Further investigation is required to understand why some real-world benchmarks suffer from more interference than synthetic ones – which are modeled to capture the worst-case behaviors. When conducting interference analysis, it can be misleading to exclusively focus on DRAM interference. Cache events can also play a pivotal role in execution time increase. In this section, we thoroughly analyze and quantify the effects of cache interference, providing a correlation between the traffic generated by the *interfering tasks*, and their effect on the memory performance of our evaluation setup. Cache interference has been addressed in the literature using techniques like *shared cache partitioning* [110, 126]. 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.

Evaluation Setup

For this evaluation, we profile the execution of the *READ_MISS* synthetic benchmark acting as a *task under test*, under different interference configurations. We focus on this benchmark because our goal for this test is to study the behaviour of the cache memory, and *READ_MISS* is the benchmark that generates regular cache memory traffic. Therefore, it can be easily tracked and analyzed. The other benchmarks (i.e., *WRITE_MISS* and *MEMSET*), generate a memory traffic that either pollutes the cache (i.e., *WRITE_MISS*), making it difficult to analyze the cache effects, or does not access the cache memory at all (i.e., *MEMSET*, see Section 4.1).

During the evaluation, we employ a reconfigurable version of *READ_MISS*, allowing us to specifically configure the memory footprint (fp) of the benchmark. The idea is to use memory footprints that are, respectively, smaller and larger than the LLC size in both setups. This way, we have both traffics that exclusively target the cache (when fp is smaller than the LLC size), and traffics that also target main memory (when fp is larger than the LLC size). 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 for the *task under test* (i.e., *READ_MISS*) in the two setups. For the *Xilinx ZU9EG*, we use memory footprints of 512KB, 768KB, 2048KB, and 16384KB. For the *Nvidia TX2*, we use memory footprints of 1024KB, 1536KB, 4096KB, and 32768KB. To analyze the behavior of the *task under test* in all those configurations, we keep track of the LLC refills that it triggers using hardware Performance Monitoring Units (PMUs) [21].

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 *THR%* parameter of the *interfering tasks*. These configurations of the synthetic benchmarks require a really low amount of time to run. As such, the results reported in this section are the average of the measurements taken by doing 1000 runs for each value of *THR%* (in general, the results were within 5% of the average).

Evaluation Results

Figures 4.8 and 4.9 present the results of the experiments for the *Nvidia TX2* and the *Xilinx ZU9EG*, respectively. In both figures, the subplots are organized in a grid. The rows report the type of traffic used by the *interfering tasks*. The columns indicate two different metrics measured on the *task under test*: the leftmost column reports the slowdown, while the rightmost one reports the number of LLC refills. Each subplot presents the results as a function of the *THR%* parameter of the *interfering tasks*. In the following, we discuss the results for each configuration of the *interfering tasks* (each row of the grid).

***READ_MISS* interfered by *READ_MISS*.** Subplots 4.8a, 4.8b, and 4.9a, 4.9b show the results obtained by using *READ_MISS* as *interfering tasks* on the *Nvidia TX2* and the *Xilinx ZU9EG*, respectively. As expected, the results heavily depend on the *fp* configuration of the *task under test*.

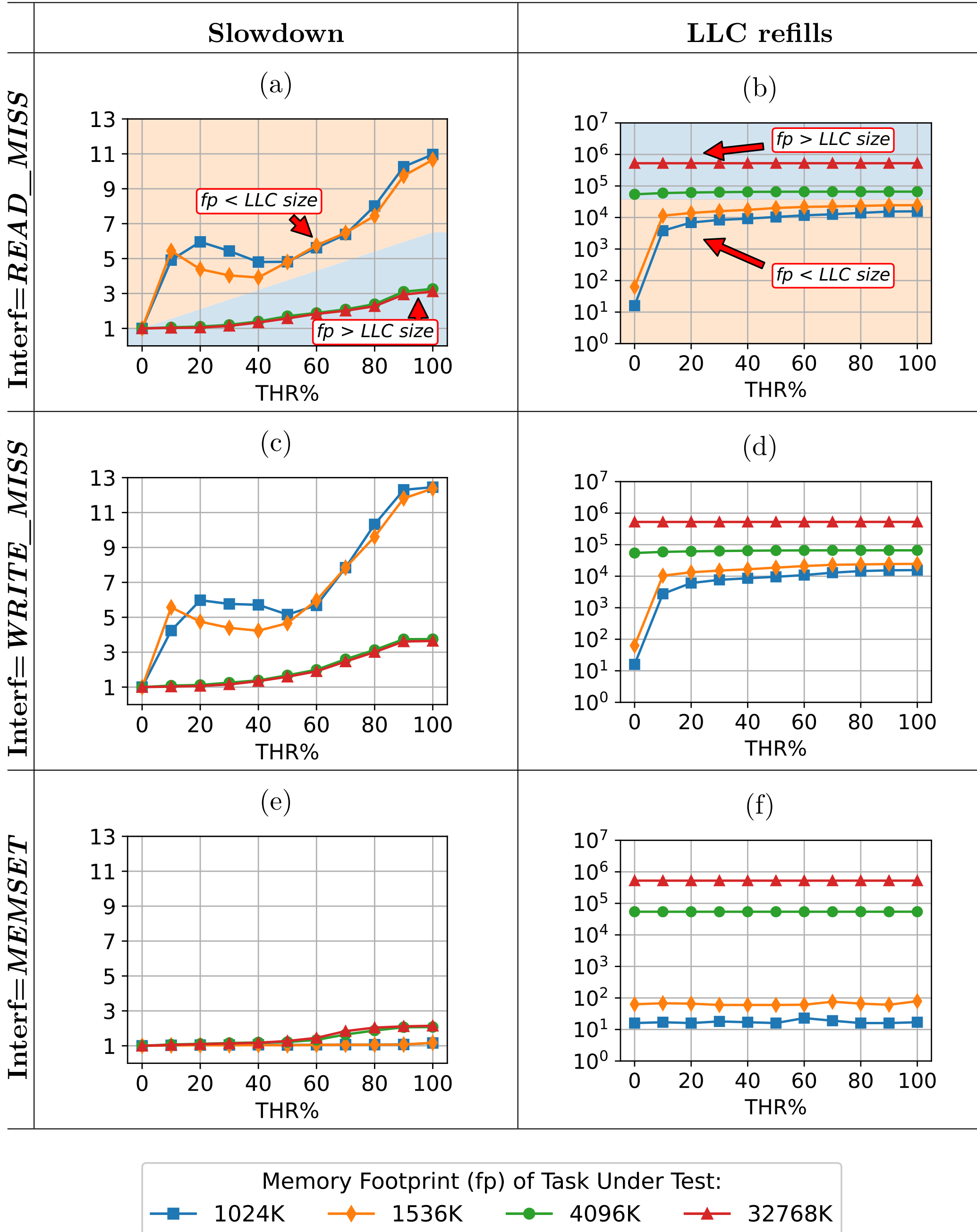


Figure 4.8: Slowdown and LLC refills caused by Cache Thrashing and Main Memory Interference on ***Nvidia TX2***

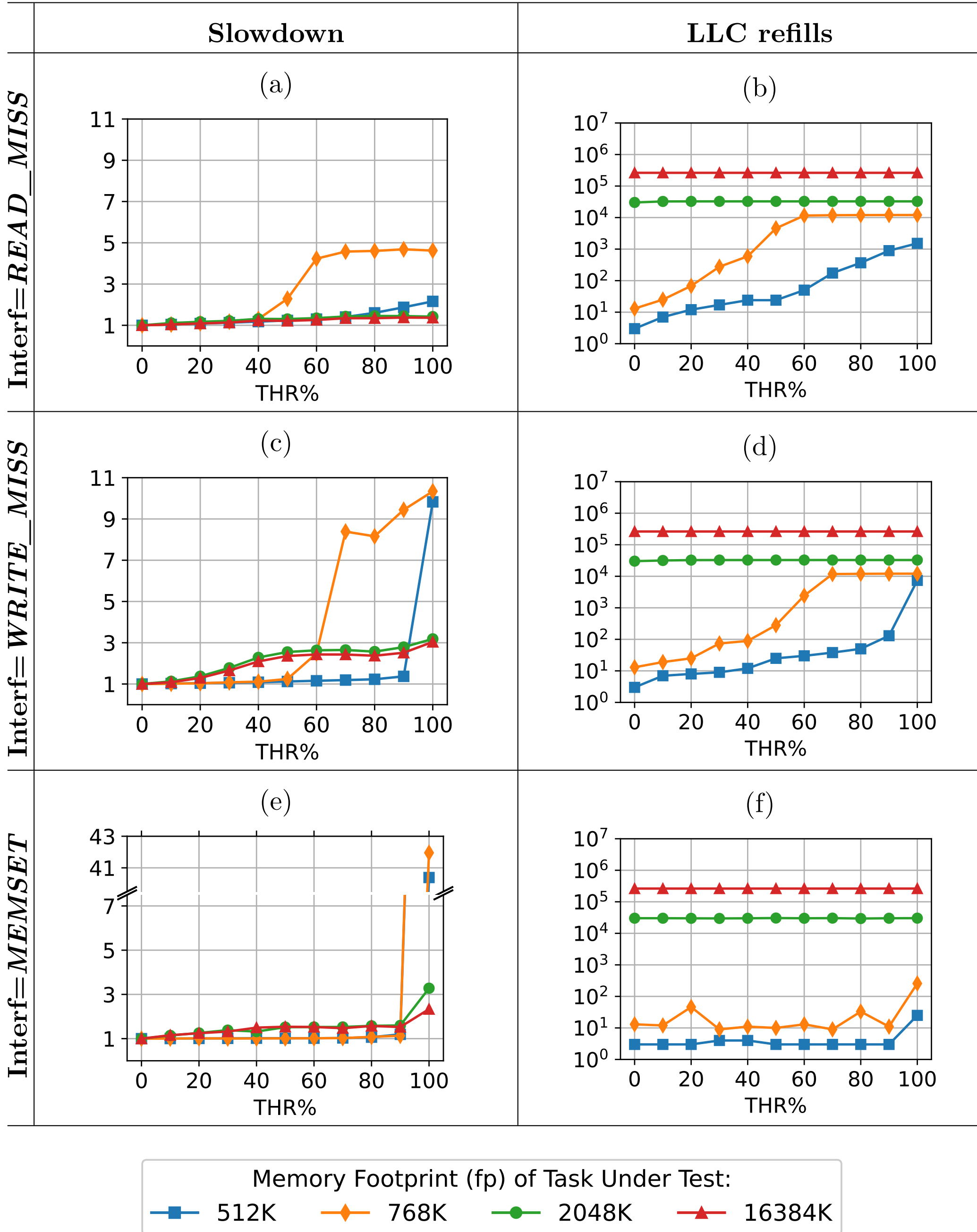


Figure 4.9: Slowdown and LLC refills caused by Cache Thrashing and Main Memory Interference on ***Xilinx ZU9EG***

If the *fp* of the *task under test* is larger than the LLC size, each *load* operation performed triggers an LLC refill, regardless of the *THR%* setting used for the *interfering tasks*. This is highlighted in Figure 4.8b for *fp* configurations of 4096K and 32768K, and Figure 4.9b for configurations of 2048K and 16384K. As a result, the number of LLC refills is constant, and the slowdown shown in Figures 4.8a and 4.9a 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% *THR%*), the memory buffer used by the *task under test* is entirely cached, and the benchmark generates a negligible amount of LLC refills. This is reported in Figure 4.8b for *fp* configurations of 1024K and 1536K, and in Figure 4.9b for configurations of 512K and 768K. As the *THR%* grows, we observe an increasing number of LLC refills due to the cache space contention generated by the *interfering tasks*. In this case, the slowdown is caused by both the DRAM interference and the LLC eviction, and therefore, it is much more severe. Ultimately, the results report a slowdown factor of 11× on the *Nvidia TX2* platform (Figure 4.9a), and 4.5× on the *Xilinx ZU9EG* (Figure 4.9a).

***READ_MISS* interfered by *WRITE_MISS*.** Subplots 4.8c, 4.8d, and 4.9c, 4.9d show the results obtained by using *WRITE_MISS* as *interfering tasks* on the *Nvidia TX2* and the *Xilinx ZU9EG*, respectively. In this configuration, the results reported in Figures 4.8d and 4.9d show that the LLC refills are comparable to the ones registered in the *READ_MISS interfered by READ_MISS* configuration (Figures 4.8b and 4.9b). However, the slowdown results (Figures 4.8c, 4.9c) are sensibly higher, especially on the *Xilinx ZU9EG* platform. To explain such a difference we repeat the benchmark execution using PMUs to profile also the number of LLC writebacks performed by the *task under test*.

The LLC writeback results are reported in Figure 4.10. When the *interfering tasks* enforce *READ_MISS* interference (Figures 4.10a, 4.10b), the number of LLC writebacks is less than 10^3 for each *fp* configuration that is used. In the same configuration, the number of LLC refills (see Figures 4.8b,

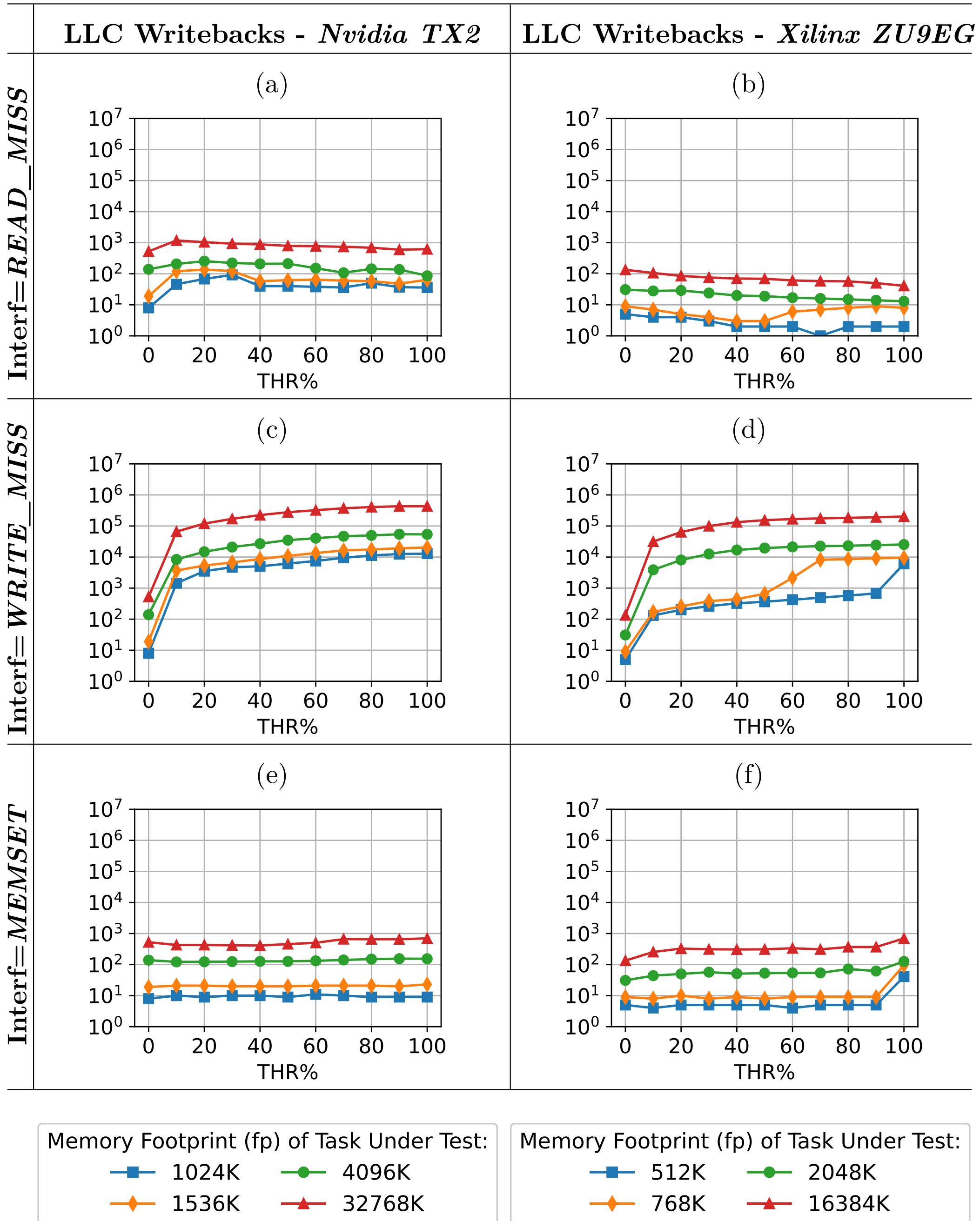


Figure 4.10: ***Nvidia TX2* (a,c,e), *Xilinx ZU9EG* (b,d,f)**. LLC Writebacks caused by Cache Trashing and Main Memory Interference

4.9b) reaches $10^5 - 10^6$. Therefore, the number of LLC writebacks represents less than 1% of the total number of main memory transactions (LLC refills + writebacks), and can be considered as negligible. This is because the traffic generated by both the *task under test* and the *interfering tasks* is read-only, and no cache line is written back to DRAM memory.

On the other hand, the number of LLC writebacks registered in the *READ_MISS interfered by WRITE_MISS* configuration (Figures 4.10c, 4.10d) grows accordingly to the *THR%* parameter of the *interfering tasks*. This happens because in both platforms dirty cached data (*dirty_bit* = 1, see Section 4.1.2) is evicted to memory only when the CPU cores try to access the corresponding cache line. In particular, the *task under test*, generating read-only traffic, triggers a number of LLC refills that evict (i.e., write back to main memory) the cache lines dirtied by the *interfering tasks*. Ultimately, this leads to an increase in the number of LLC writebacks transactions over the *worst-case* registered by the *READ_MISS interfered by READ_MISS* configuration, and therefore to a higher slowdown in the benchmark execution time. This configuration represents the worst-case slowdown scenario for the *Nvidia TX2* platform, as the reported slowdown reaches up to a factor 12.5×. Whereas, on the *Xilinx ZU9EG*, we register a 10× increase in execution time.

***READ_MISS* interfered by *MEMSET*.** As explained in Section 4.1, the traffic generated by *MEMSET interfering tasks* is 100% *write-no-allocate*. This write-only traffic directly writes out to main memory 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 4.8e, 4.8f, 4.10e, and 4.9e, 4.9f, 4.10f demonstrate this aspect. In fact, the impact of the *THR%* parameter on the number of LLC refills/writebacks is almost null or negligible if compared to other memory interference configurations (*READ_MISS* vs *READ_MISS*, or *READ_MISS* vs *WRITE_MISS*). Nevertheless, the slowdown results are very different between the two setups. On the *Nvidia TX2* (Figure 4.8e), 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 4.9e), we obtain a 2.5×-3.5× slowdown when the *fp* is larger than the LLC size, which is comparable to the one obtained using *WRITE_MISS* interference, and an unexpected **42**× slowdown when the *fp* is smaller than the LLC size. To understand the cause of this slowdown, we better analyze the cache 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 [73]. 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 [11]. 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 [22]. This causes the CPU cores to stall until the entries in the store buffer are freed (written back to memory).

Cache blocking can happen in the *READ_MISS* interfered by *MEMSET* scenario: the *write-no-allocate* traffic generated by the *MEMSET interfering tasks* can saturate the store buffer, causing the LLC cache to block. This, in turn, can cause the CPU core executing the *READ_MISS* task to stall. To demonstrate this phenomenon, we repeat the execution of the *task under test* (*fp* = 512KB) both in isolation and co-scheduled with *MEMSET interfering tasks*. During the execution, we use PMUs to profile the execution of both the *task under test* and the *interfering tasks*. For the *task under test*, we configure PMUs to track the *CPU stalls because of load misses* event (PMU event number 0xE7). For the *interfering tasks*, we configure 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 4.11. In the figure, the subplots present the results as a function of the *THR%* parameter of the *interfering tasks*. As shown in Subplot 4.11b, when the *THR%* parameter changes from 90% to 100%, the number of *WRITE*

READ_MISS* (fp = 512KB) interfered by *MEMSET

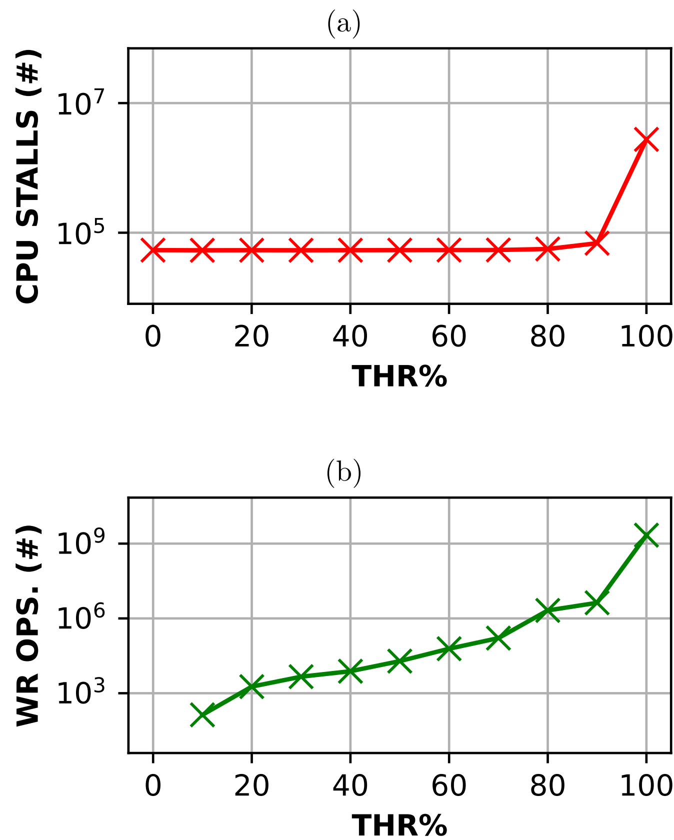


Figure 4.11: *READ_MISS* (fp = 512KB) interfered by *MEMSET*. The *CPU STALLS because of load miss* (Subplot 4.11a) are measured on the *task under test*. The *WRITE OPERATIONS that stall the pipeline because the store buffer is full* (Subplot 4.11b) are measured on the *interfering tasks*

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 4.11a) grows from $\approx 10^5$ to $\approx 10^7$. This result indicates that when the *THR%* parameter reaches 100%, the store buffer gets saturated by the memory requests of the *MEMSET interfering tasks*, causing the CPU core executing the *task under test* to stall, and ultimately, the observed slowdown.

Analysis Summary

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

4.3 Discussion and Conclusion

In this chapter, we presented a thorough analysis of main memory and cache interference effects on two representative multicore SoCs: the *Nvidia TX2* and the *Xilinx ZU9EG*. We show that main-memory bound read-only and write-only traffics generated by synthetic workloads don't represent worst-case in terms of sensitivity to memory interference. In contrast, we show that different workloads generate different levels of interference and that there is no single traffic type that generates the highest amount of interference on all platforms.

We confirm that, to find the worst-case for the hardware under analysis, an in-depth interference characterization is necessary. On our setup, we highlighted that the impact of interference depends on the memory hierarchy level where it occurs: we observed up to $3\times$ slowdown due to main memory interference, up to $13\times$ slowdown due to the combination of cache interference and main memory interference, and a $42\times$ slowdown due to an architectural bottleneck in the memory hierarchy of the *Xilinx ZU9EG*.

There are multiple ways in which current and future works may be affected by our findings and from the methodology we used in this chapter. Memory interference mitigation techniques [124, 121, 130, 108], for example, may tailor their mitigation strategies on the characterization introduced in this chapter. This improved analysis could lead to a better estimation of the interference that a platform is subject to at runtime. This, in turn, would allow the design of more precise and less over-conservative approaches, reducing bandwidth regulation inefficiency [40]. Similar considerations apply to those approaches that perform formal schedulability analyses [13, 4, 14]. The thorough interference characterization presented in this work could lead to better WCET estimations in these works, leading to less conservative results. This applies to approaches based on both formal analysis [4] and heuristics [14]. Lastly, the methodology discussed in this work can lay the base for future interference characterizations of future platforms. Systematic approaches to perform memory interference characterization [109] can build upon the methodology presented in this work to evaluate a wide spectrum of architectural phenomena.

Chapter 5

Synchronous vs. Asynchronous Reconfiguration of Memory Bandwidth in MBMSs: a Comparative Analysis

The high degree of parallelism employed in modern HeSoCs introduces challenges when it comes to predictably executing multiple tasks in parallel, due to the high degree of physical resource sharing. Main memory sharing, in particular, has been shown to represent a significant bottleneck that may lead to a reduction in the bandwidth available to individual cores, which ultimately translates into a relevant slowdown of the software tasks and an overall unpredictable system behavior [80].

The scientific community proposed several techniques to tackle this challenge at both the hardware and software level [80]. Among those techniques, *bandwidth regulation* allows to control the amount of memory bandwidth available to a specific core at any given time [124, 130, 105], enabling disciplined bandwidth usage from low-priority tasks and ultimately ensuring that highly-priority ones can meet their timing constraints. However, modern real-time applications exhibit significant dynamism, with a varying number of tasks being co-scheduled at any moment in the system. Internally, those

tasks alternate between compute-intensive and memory-intensive phases, the latter exhibiting different memory access patterns, which can be more or less prone to contributing to the memory interference problem. The granularity at which time-critical activities can be identified at the core level can thus be way finer than a whole task, and just span smaller *memory-critical section* (MCS) of the task itself. MCS are highly sensitive (and greatly contribute) to interference, and constitute the only portion of a dynamic workload that should be disciplined by techniques such as *bandwidth regulation*. Core- or task-level *bandwidth regulation* is thus too coarse-grained and too static in this respect, and leads to suboptimal use of the system bandwidth.

For any system-level *Memory Bandwidth Management Scheme* (MBMS) to be capable of dynamic, fine-grained bandwidth regulation, more or less frequent *reconfiguration* of its parameters is key. In particular, the overhead and latency of such *reconfiguration techniques* determine the feasibility of the overall MBMS, defining its precision in adapting to dynamic system changes in terms of memory usage. Dynamic reconfiguration of bandwidth regulation techniques has been studied to some extent both in homogeneous [130, 121] and heterogeneous [30, 104] systems. Yet, there has been limited research in systematically evaluating the overheads and latencies associated with the *reconfiguration* step of a MBMS.

In this chapter, we address this gap by presenting a comparative analysis of the two basic techniques for reconfiguring bandwidth regulators on a multicore CPU: *synchronous* and *asynchronous*. These techniques adopt opposing approaches to detect bandwidth reconfiguration points. Specifically, *synchronous* uses periodic interrupts, guaranteeing fixed latency. In contrast, *asynchronous* relies on task-triggered interrupts, offering lower latency and potentially lower overhead. Both techniques are implemented in a software-based MBMS based on MCSs. We evaluate their impact by studying the effects of their latency and overhead on i) the time granularity at which the MBMS can successfully control the MCS timing under interference and on ii) the ability of the MBMS to efficiently use the overall system memory bandwidth. We conduct the experiments on a real-world setup, a *Xilinx ZU9EG* platform. The results, obtained through synthetic and real-world

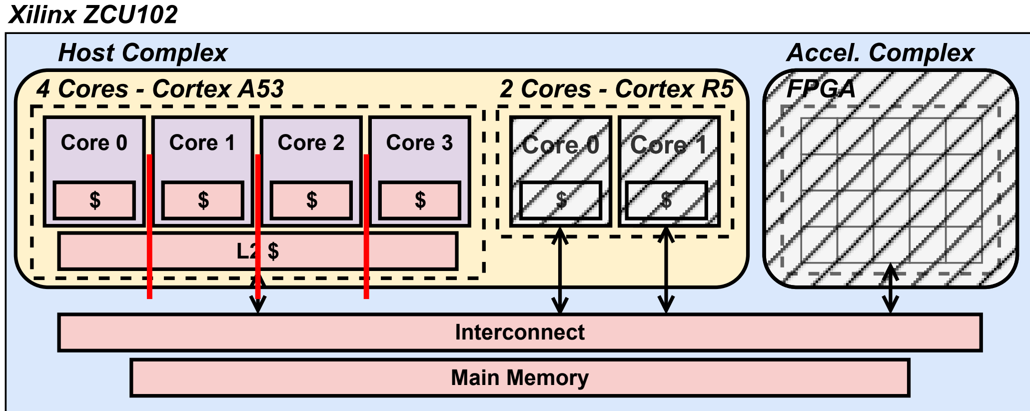


Figure 5.1: *Xilinx ZU9EG* architecture.

benchmarking (the *PolyBench* suite [95]), show that the *asynchronous* technique improves control granularity of the *synchronous* one by up to a factor of 19x, reducing latency from the *ms* to the μs scale.

5.1 Hardware and Software Model

Figure 5.1 shows the architectural model of the HeSoC we use in this analysis: a *Xilinx ZU9EG* platform. Also in this analysis, we focus on the *host*-complex. Therefore, in the figure, the *Accelerator* complex and the Host-level cores exceeding the main *Application Processing Unit* are *grayed out*. Moreover, as our focus is on main memory interference, we assume that caches are either physically private to every core, or made private-like via cache partitioning/coloring techniques [122, 118]. In this model, memory interference occurs exclusively at system interconnect and main memory level. We refer to this interference as *main-memory interference*. A detailed description of the hardware components of the *Xilinx ZU9EG* platform is reported in Section 5.4.

We employ a task model based on *Memory-Critical Sections* (MCSs), code portions in tasks highly sensitive to memory interference. In the model we adopt, MCSs represent the time-critical portions of tasks: they have to meet *Quality of Service* (QoS) requirements in terms of maximum tolerated

slowdown. To meet QoS requirements, MCSs are executed at maximum privilege level in the system (i.e., they are not subject to preemption or CPU migration). However, being sensitive to memory interference, they still may suffer unpredictable slowdowns due to the memory interference generated by co-scheduled tasks. This, in turn, would break the QoS requirements.

To address the memory interference, we adopt a MBMS to protect MCSs: during MCSs, the MBMS eliminates memory interference by *regulating* the memory bandwidth of out-of-MCSs tasks. Specifically, this MBMS uses *reconfiguration* techniques to detect the start and finish of MCSs, which in this model represent the bandwidth *reconfiguration* points. Programmers explicitly mark MCSs start/finish within tasks'code using an API ($mcs_start()$ and $mcs_finish()$ calls), which is provided by the MBMS.

This model requires the highest possible level of dynamism, as *reconfiguration* of memory bandwidth happens at MCS level, inside tasks. This entails higher dynamism when compared to, for instance, CPU-core level [124] or task level [51] dynamism. Thus, it enables us to finely evaluate the *reconfiguration* techniques in a MBMS.

The evaluation we propose is built upon synthetic and real-world workloads. In both setups, the taskset consists of one task per CPU core. Specifically, the taskset includes a single task executing MCSs, referred to as the *main* task, and other tasks (one per remaining CPU core) that generate memory interference. We refer to those tasks as *other* tasks. By having only one task per CPU core, we eliminate scheduling effects (and the related overhead). As a result, the primary source of interference affecting the *main* task is the memory interference generated by the *other* tasks. This ensures clean and non-noisy results.

In the evaluation, we discuss the *slowdown* due to memory interference in the form of percentage, i.e., the percentage increase in execution time due to memory interference (e.g., 10% *slowdown* corresponds to 1.1× execution time).

5.2 Memory Bandwidth Management Schemes

In the analysis, we compare two different MBMSs based on MCSs. This means that the MBMSs mitigate the memory interference generated by *other* tasks only during the MCSs executed by the *main* task. The two variants of MBMS leverage the two basic techniques to implement *reconfiguration*: *synchronous* and *asynchronous*. For clarity, we refer to the resulting MBMSs by the names of their respective *reconfiguration* techniques (i.e., *SYNCHRONOUS* and *ASYNCHRONOUS*).

Below, we first explain our specific choice for the bandwidth *regulation* components of our MBMSs. Then, we discuss the design and implementation of the *reconfiguration* techniques used by the MBMSs, highlighting their respective pros and cons.

5.2.1 Bandwidth Regulation

In this analysis, we evaluate MBMSs by examining two key performance metrics: i) timing granularity, which represents the minimum MCS duration that a MBMS can effectively protect, and ii) overall bandwidth utilization, which reflects its ability to allocate full memory bandwidth to tasks when no task in the system is executing MCSs. These metrics depend on the MBMS's ability to identify bandwidth reconfiguration points (determined by the *reconfiguration* technique) and enforce a bandwidth regulation scheme (determined by the *regulation* technique). Therefore, both *reconfiguration* and *regulation* techniques are crucial to the performance of a MBMS. However, each technique is limited by its respective overhead and latency.

In this work, we aim at isolating the latency and overhead of *reconfiguration* techniques. Therefore, we need to eliminate the impact of the *regulation* technique on the performance of a MBMS. The cleanest, if not the only solution to eliminate this component, is to turn off the *regulation* technique in the MBMS. If no real *regulation* technique is in place, the only remaining option to control memory interference is to switch off *other* tasks during MCSs. In other words, the MBMSs we evaluate stall *other* tasks as soon as the *reconfiguration* technique detects the execution of a MCS in the system.

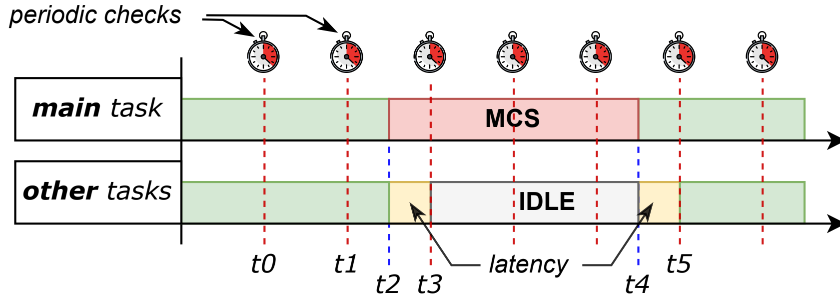


Figure 5.2: Execution model of the **synchronous** *reconfiguration* technique.

We discuss possible implications of including *regulation* techniques in Section 5.7.

5.2.2 *SYNCHRONOUS*

Figure 5.2 presents the execution model of the *synchronous reconfiguration* technique used by *SYNCHRONOUS*. From top to bottom, the figure depicts the running status of the *main* task executing a MCS, and the *other* tasks. The *synchronous reconfiguration* technique periodically checks (*t0, t1, t3 ...*) whether the *main* task is executing MCSs. During a periodic check, if the *main* task is executing a MCS (*t3*), then the *other* tasks are stalled, and in turn, the memory interference is limited. In the same way, at the end of the MCS (*t4*), the periodic check re-activates *other* tasks (*t5*), and full memory bandwidth is restored for each task in the system.

This technique can be configured using a tuning parameter, the *reconfiguration period* (also referred to as *period*). This parameter defines the time interval used for the periodic checks (*t1 - t0*), and in turn, it also determines the control granularity of the technique. In fact, the closer the periodic checks are, the more the *reconfiguration* technique is precise in detecting the beginning/end of the MCSs (i.e., *latency, t3-t2*), leading to more precise control of memory bandwidth. However, this parameter also determines the overhead, as the periodic checks interrupt the execution of both *main* and *other* tasks. Thus, a comprehensive analysis of the optimal value for this parameter must be conducted to achieve the highest possible control granularity while

Algorithm 3: *SYNCHRONOUS*

```
1  struct task_struct { ... int mcs_val; ... }
2  Function nr_mcs_cores():
3      nr_mcs_cores ← 0
4      for core in cpu_cores() do
5          if current_task_of_core(core) → mcs_val == 1 then
6              nr_mcs_cores ← nr_mcs_cores + 1
7          end
8      end
9      return nr_mcs_cores
10 Function timer_irq_handler():
11     curr ← current_task()
12     if is_mcs_requested_by(curr) then
13         curr → mcs_val ← 1
14     else
15         curr → mcs_val ← 0
16     end
17     if nr_mcs_cores() > 0 and curr → mcs_val == 0 then
18         activate(kthrottle)
19     else
20         deactivate(kthrottle)
21     end
```

maintaining a feasible overhead.

Implementation

Our implementation of *SYNCHRONOUS* comprises a kernel module and a user-space library. The kernel module's structure is reported in Algorithm 3. Upon initialization, the kernel module allocates per-core shared memory pages. These pages are used by the *main* task to signal the execution of a MCS. When the *main* task initializes the user-space library, it maps a portion of the kernel's shared memory page to its own memory space. Note that it is a per-core memory page. As a result, since tasks are pinned to CPU cores, they directly write (via $mcs_start()$ and $mcs_finish()$ calls) the memory-mapped region to signal the execution of MCSs. The use of shared pages

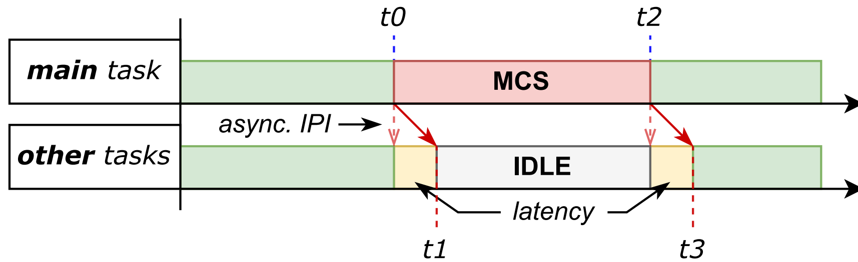


Figure 5.3: Execution model of the **asynchronous** *reconfiguration* technique.

enables us to eliminate the need for system calls to signal the execution of MCSs, reducing overhead. Furthermore, the use of multiple per-core memory pages instead of a single shared page enable cores to access data structures independently, i.e., without synchronization. This further reduces overhead.

The periodic checks are implemented as a periodic timer event (Line 10). During each occurrence, the timer event handler checks whether the current task is executing a MCS by searching its core's shared memory page (Line 12). Then, it stores the obtained value in the task control block structure (Lines 13, 15), which we modified to account for whether a task is executing a MCS (Line 1). Subsequently, it counts the number of tasks executing MCSs (Line 17) and eventually activates a high-priority idle thread, i.e., *kthrottle* (Lines 18, 20). When active, *kthrottle* preempts the current task and nullifies the memory interference generated by the CPU core.

5.2.3 *ASYNCHRONOUS*

Figure 5.3 presents the execution model of the *asynchronous reconfiguration* technique used by *ASYNCHRONOUS*. This technique is based on asynchronous inter-processor interrupts (IPIs), which are broadcast by the *main* task when it enters or exits MCSs (*t0, t2*). Upon receiving such IPI, each core checks whether the *main* task is executing a MCSs (*t1, t3*). If so, then the core is stalled (*t1*), otherwise it continues execution (*t3*).

Algorithm 4: *ASYNCHRONOUS*

```
1  mcs_cores_bitmask ← {0, 0, ...}        // protected cores bitmask
2  Function update_cpu_status():
3  |  cpu ← this_cpu()
4  |  if bitmask_empty(mcs_cores_bitmask) or
   |   bitmask_test(mcs_cores_bitmask, cpu) then
5  |  |  deactivate(kthrottle)
6  |  else
7  |  |  activate(kthrottle)
8  |  end
9  Function mcs_set_priv(priv):
10 |  cpu ← this_cpu()
11 |  bitmask_set(mcs_cores_bitmask, cpu, priv)
12 |  on_each_cpu(update_cpu_status)          // Asynch. IPI
```

Implementation

ASYNCHRONOUS is composed of a kernel module and a user-space library. Algorithm 4 reports the structure of the kernel module. Upon initialization, the kernel module allocates a shared bitmask (*mcs_cores_bitmask*, Line 1) that contains the running status of each core, i.e. whether the cores are executing MCSs or not. Initially, this bitmask is empty. The *mcs_start*() and *mcs_finish*() primitives are implemented via system calls (*mcs_set_priv*(*priv*), Line 9). When the *main* task executes these system calls, it sets its running state into the *mcs_cores_bitmask* (Line 11) and sends an asynchronous IPI to all cores in the system to notify an update (Line 12). The asynchronous IPI is handled on the receiving cores by executing the *update_cpu_status*() function (Line 2). Note that this function is also executed by the calling core. This function either activates or deactivates a high-priority idle thread (i.e., *kthottle*) to stall the receiving core depending on the status of the *mcs_cores_bitmask*. Specifically, if no cores in the system or the executing core itself is executing a MCS (Line 4), then *kthrottle* is deactivated. Otherwise, *kthrottle* is activated, and the task scheduled on the receiving core is preempted. In turn, the memory interference generated by the core is nullified.

5.2.4 Discussion

Both MBMSs exhibit intrinsic limitations due to the *reconfiguration* technique they leverage. In *SYNCHRONOUS*, the *reconfiguration period* defines the control granularity of the MBMS, as MCSs that are shorter than the *reconfiguration period* may not be detected at all. On the other hand, *ASYNCHRONOUS* is limited by both the use of system calls to enter/exit MCSs, which are costly, and the overhead and latency due to the asynchronous IPI (Figure 5.3, *t1-t0*). As a result, both the duration and the interleaving time of the MCSs can exceed the minimum control granularity of the *reconfiguration* techniques used by the MBMSs, resulting in unprotected MCSs or overall bandwidth underutilization, respectively. Our analysis provides us with a quantitative evaluation of these limits.

5.3 Synthetic benchmark

This section presents the synthetic benchmark we employ in our evaluation, which enables us to assess the performance of the MBMSs in a wide range of MCSs execution scenarios. The synthetic benchmark comprises a single *main* task executing MCSs, co-scheduled with *other* tasks (one for each remaining CPU core) generating memory interference. Below, we detail the characteristics of both types of tasks.

5.3.1 Main task

main is a custom memory-intensive task based on the `bandwidth-rt` benchmark, from the *IsolBench* suite [65]. Its structure, reported in Algorithm 5, is based on a loop that iteratively alternates memory-intensive code sections (i.e., MCSs) and idle sections. Both sections are configurable via two parameters:

- *MCS_size*, which sets the number of memory operations performed in each MCS.

- *duty_cycle*, which determines the ratio between the duration of each MCS and the respective iteration duration.

Intuitively, *MCS_size* determines the duration of the MCSs, while *duty_cycle* is used to compute the MCSs interleaving time. Thanks to its structure and tunability, *main* allows us to generate various MCSs patterns (i.e., short/long, sparse/frequent MCSs), useful to stress the control granularity of the *reconfiguration* techniques under analysis.

At runtime, during each iteration, *main* times the execution of a MCS (lines 17-23) and subsequently sleeps for a time calculated based on the previously measured MCS duration and the desired *duty_cycle* (line 25). In particular, each MCS is composed of three steps: i) MBMS activation (line 19). ii) Execution of *MCS_size* memory operations (line 20). iii) MBMS deactivation (Line 21). After a predefined number of iterations (i.e., *iter*, line 16), *main* returns: *MCS_list*, which is a list containing the MCSs durations, and *tot_exec_time*, indicating the total execution time of the benchmark.

5.3.2 Other tasks

other tasks continuously spin on a memory-intensive loop to maximize the memory interference on the *main* task. During each iteration, *other* tasks execute a block of memory operations moving *ch_size* bytes to/from memory (Line 7), and keep track of the number of bytes transferred (Lines 8, 9). In particular, they account for both the total number of bytes (*bytes_tot*, Line 8) and the number of bytes transferred during the *main* task MCSs (*bytes_in_MCS*, Line 9).

To keep track of the bytes transferred during the *main* task MCSs, *other* tasks use a global flag (*main_in_mcs*, Line 9), which indicates whether the *main* task is executing a MCS. The value of *main_in_mcs* is set/unset by *main* at the beginning/end of each MCS (Lines 18, 22).

The execution of *main* and *other* tasks is synchronized using a software barrier (Lines 5, 14). This means that none of the tasks start until all of them have reached the barrier. However, there is minimal delay between the start of tasks: we empirically measure an 80ns delay between the beginning

Algorithm 5: Synthetic benchmark

```
1  main_in_mcs ← 0  // Global flag, indicates if main is in MCS
2  Function other(ch_size) :
3      bytes_tot ← 0                          // Mem. bytes counter
4      bytes_in_MCS ← 0     // Mem. bytes counter during MCSs
5      synchronize()                          // Synch. with main
6      while main_running()  do
7          mem_ops(ch_size)
8          bytes_tot += ch_size
9          bytes_in_MCS += ch_size * main_in_mcs
10     end
11     return bytes_in_MCS, bytes_tot
12 Function main(iter, MCS_size, duty_cycle) :
13     MCS_list ← list()
14     synchronize()                   // Synch. with other tasks
15     global_start ← time()
16     for i ← 0 to iter  do
17         MCS_start ← time()
18         main_in_mcs ← 1
19         mcs_start()                             // MCS start
20         mem_ops(MCS_size)
21         mcs_finish()                            // MCS finish
22         main_in_mcs ← 0
23         MCS_dur ← time() −MCS_start
24         append(MCS_list, MCS_dur)
25         sleep(MCS_dur/duty_cycle − MCS_dur)
26     end
27     tot_exec_time ← time() −global_start      // Tot. execution time
28     return MCS_list, tot_exec_time
```

of the first MCS executed by *main* and the beginning of *other* tasks. For the configurations of *main* used in the evaluation (see Section 5.5.2), this delay accounts for less than 1% duration of the first out of 1000 MCSs executed. Thus, we consider it negligible.

other tasks continue running until *main* completes its last iteration (Line 6). As a result, they generate memory interference for the entire duration of *main*.

5.3.3 Output

The output of the synthetic benchmark is composed of three metrics:

1. The list of the *main* task MCS durations (*MCS_list*, Line 28).

2. The average memory bandwidth perceived by the *other* tasks **inside** the *main* task MCSs, computed as:

$$\frac{\sum_{i=0}^{n}(bytes_in_MCS_i)}{(n) * \sum_{j=0}^{iter}(MCS_list_j)} \tag{5.1}$$

3. The average memory bandwidth perceived by the *other* tasks **outside** the *main* task MCSs, computed as:

$$\frac{\sum_{i=0}^{n}(bytes_tot_i - bytes_in_MCS_i)}{(n) * (tot_exec_time - \sum_{j=0}^{iter}(MCS_list_j))} \tag{5.2}$$

We refer to these metrics as ***MCS durations***, ***bandwidth inside*** MCSs, and ***bandwidth outside*** MCSs, respectively.

5.3.4 Running configurations

We execute the synthetic benchmark in three running configurations:

1. ***solo***, where *other* tasks are not scheduled, and the *main* task executes without memory interference.

2. ***regulated***, where *other* tasks are scheduled, and the MCSs are protected using the MBMSs.

3. ***unregulated***, where the $mcs_start()$ and $mcs_finish()$ functions in *main* are mapped to `nop` operations, and the MCSs execute with uncontrolled memory interference (i.e., no MBMS activation).

The *solo* and *unregulated* configurations serve as baselines of no memory interference and uncontrolled memory interference, respectively. The *regulated* running configuration indicates the effectiveness of a MBMS in both memory interference mitigation during MCSs and bandwidth re-allocation outside MCSs. Note that *solo* and *unregulated* configurations are independent of the MBMSs, while the results obtained in *regulated* running configuration depend on the MBMS in use. By comparing the result obtained in *regulated* configuration across multiple MBMSs, we compare the performance of the respective *reconfiguration* techniques, which is central to this work.

5.4 Evaluation Setup

This section presents the hardware and software setup, the QoS requirement of the MCSs, and the *period* configurations of *SYNCHRONOUS* we employ in the evaluation.

5.4.1 Hardware and software setup

We perform the analysis on a *Xilinx Zynq UltraScale+ (ZCU102)* platform based on a quad-core Cortex-A53 CPU. In this platform, each CPU core has a private 32KB L1I cache and a 32KB L1D cache, while all cores share a 1MB L2D cache. The CPU cores are connected to a 4GB DRAM memory module through a shared 128 bit-wide interconnect.

We implement and evaluate the two MBMSs in a Linux 5.10 kernel (PetaLinux). For the evaluation, we disable CPU frequency scaling by setting the CPU frequency governor to *performance*. A prior work in the

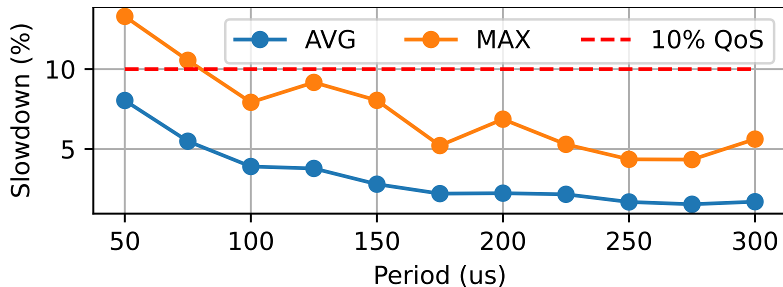


Figure 5.4: Slowdown induced by the overhead of *SYNCHRONOUS* as a function of its *period* configuration.

literature introduced BWLOCK [121], a MBMS based on MCSs that employs a *synchronous reconfiguration* technique. Our implementation of *SYNCHRONOUS* represents a porting of BWLOCK on our reference setup.

5.4.2 QoS slowdown threshold

We evaluate the MBMSs by comparing the respective control granularities, i.e., the minimum MCS durations required to mitigate the slowdown effect caused by memory interference. To achieve this, we define a maximum slowdown threshold tolerated by the MCSs and evaluate the ability of the MBMSs to reduce the slowdown suffered by MCSs within this threshold. Specifically, we select a 10% slowdown threshold, which is used in previous literature [112]. For better clarity, we name it δ_{10}.

5.4.3 *SYNCHRONOUS period* parameter configuration

As explained in Section 5.2.2, in *SYNCHRONOUS*, the *period* parameter defines the control granularity of the MBMS. The shorter the period, the finer the granularity, but the higher the overhead, as the MBMS code periodically interrupts the tasks' code.

Based on empirical evaluation, on our setup, the overhead of the periodic checks averages $4\mu s$, with a peak value of $9\mu s$ (per period). This overhead eats out parts of the tolerated slowdown identified by the slowdown threshold:

given the latter, the finest-granularity option is to choose the shortest period that induces a slowdown lower than the threshold. Larger periods can be chosen to achieve smaller overheads at the cost of less precise *reconfiguration*.

We estimate the optimal *reconfiguration period* by evaluating the slowdown suffered by a taskset when the kernel module of *SYNCHRONOUS* is active in the system. Specifically, we execute tasks from the *PolyBench* [95] suite. Figure 5.4 shows the average and peak slowdown recorded by these tasks as a function of the *reconfiguration period*. Specifically, the figure shows that at $100\mu s$ period, the MBMS induces $\approx 4\%$ average and $\approx 8.5\%$ peak slowdown, which is consistent with the $4\mu s$ and $9\mu s$ overhead that we measured. According to the figure, the minimum *period* to meet our target 10% slowdown threshold is $100\mu s$. To include lower-overhead configurations of the MBMS, we also select $250\mu s$, $500\mu s$, and $1000\mu s$ configurations.

5.5 Synthetic Benchmark Evaluation

In this section, we describe the evaluation and the results obtained using the synthetic benchmark (see Section 5.3). This analysis enables us to assess the control granularity of the two MBMSs, evaluating a wide range of MCSs durations and interleaving times scenarios. Below, we present the evaluation process, the synthetic benchmark configuration used in the evaluation, and the results.

5.5.1 Evaluation Process

The evaluation involves multiple executions of the benchmark, varying the parameters configuration of *main* (i.e., *MCS_size* and *duty_cycle*), the running configuration (i.e., *solo*, *unregulated*, and *regulated*), and the type of the MBMS in use. First, for each combination of the *main* task parameters, the benchmark is executed in *solo* and *unregulated* configurations, without the intervention of any MBMS. Next, for each combination of *main* task parameters and MBMSs, the benchmark is executed in *regulated* configuration. By comparing the results obtained in *regulated* configurations across

multiple MBMSs, we can compare the control granularity of the respective *reconfiguration* techniques.

5.5.2 Synthetic Benchmark Configuration

As stated in Section 5.3, *main* can be tuned via two parameters: *duty_cycle* and *MCS_size*. We configure it to use *duty_cycle* configurations of 10%, 50%, and 90%, representing a wide range of MCS sparsity scenarios (i.e., sparse to frequent MCSs). We vary the *MCS_size* parameter setting in the range $[2^8 \rightarrow 2^{17}]$, stepping by powers of two (i.e., $2^8, 2^9, ..., 2^{17}$). Empirical evaluation on our platform shows this value range results in MCSs of $\approx 10\mu s$ to $\approx 10ms$. We select this range because lower values would fall below the control granularity of both MBMSs, while higher values would fail to reveal further performance differences between them. To obtain comprehensive results, *main* executes 1000 iterations. Moreover, for improved statistical significance, we repeat each benchmark execution 10 times, making a total of 10000 MCS duration samples for each combination of *MCS_size* and *duty_cycle*.

main accesses a 32MB memory buffer using a read-only strided access pattern. In particular, it performs a series of load (i.e., `ldr`) operations separated by a fixed stride of 320 bytes. The choice of this memory access pattern is motivated by several aspects: i) Thanks to the memory buffer size and the stride, each memory load operation causes an L2 cache refill, therefore a main memory access. ii) Since the memory traffic is read-only, the cache pollution (i.e., dirty cache lines) is limited, and the number of spurious L2 cache writebacks – which would cause results unstability – is minimized. iii) In our platform, this stride value (i.e., 5 cache lines) disables cache prefetchers [17]. As a result, we found this memory access pattern to be particularly sensitive to memory interference. Moreover, it is the most stable in terms of retrieved results across multiple executions of the benchmark.

other tasks generate non-cached memory traffic using the memory instruction *Data Cache Zero by Virtual Address* (i.e., `dc zva`). On our platform, this instruction clears main memory without causing L1 or L2 cache

allocations when it misses in the cache. This prevents cache space contention. Specifically, we select this type of *other* tasks because, through empirical evaluation, we observed that they generate the maximum memory interference on our setup. When the memory interference is maximized, the impact of memory interference control is also maximized. Consequently, this choice of *other* tasks enables us to highlight (maximize) the performance gap between the two *reconfiguration* techniques in the results.

The *ch_size* parameter of the *other* tasks determines the number of bytes transferred to memory during each iteration. Consequently, it also determines the duration of the iterations, and the frequency used by the *other* tasks to check whether the main task is executing a MCS. This parameter should allow *other* tasks to perform multiple iterations for each iteration of the main task. This way, they precisely account for the memory bandwidth perceived inside and outside MCSs. We empirically set its value to 256.

5.5.3 Evaluation Results

We present the results in Figures 5.5 and 5.6. Within the figures, each subplot presents an output metric of the benchmark (i.e., *MCS duration*, *bandwidth inside* MCSs, or *bandwidth outside* MCSs) as a function of both the *main* task parameters and the MBMS in use. Each subplot presents a secondary x-axis on the top, where each point indicates the average *MCS duration* in *solo* configuration that corresponds to the value of the *MCS_size* parameter in the same position on the bottom x-axis. We report both x-axes because the one at the bottom indicates the effective configuration of *main*, while the one on top enables us to evaluate the control granularity of the MBMSs in terms of MCS duration. For better clarity, we discuss the results using the x-axis on top as a reference.

We present the *MCS duration* results (Figure 5.5) as the slowdown percentage of the MCSs in *regulated* and *unregulated* configurations compared to *solo* configuration. Specifically, this slowdown is computed using the 99th percentile *MCS duration* of *regulated* and *unregulated* configurations normalized over the average *MCS duration* of *solo*. This choice is made for two

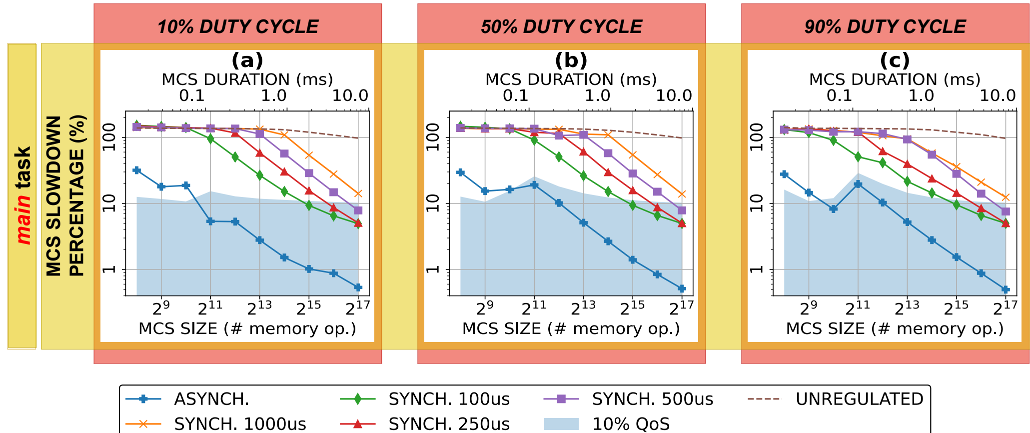


Figure 5.5: *MCS SLOWDOWN* results obtained using the synthetic benchmark.

reasons: i) to remove possible outliers, as short MCSs (i.e., $\approx 10\mu s$) show up to 300% time variability caused by the Linux operating system noise. Using the 99th percentile results reduces this time variability down to $\leq 10\%$; ii) to leverage a stable normalization baseline. For short MCSs (i.e., $\approx 10\mu s$), the 99th percentile results still exhibit up to 10% time variability compared to the average. Using them as a normalization baseline would introduce instability, affecting the reliability of the normalized results. Therefore, we choose the average value of *solo* running configuration as the normalization baseline.

The MCS slowdown results report the 10% slowdown threshold we select (i.e., δ_{10}, see Section 5.4.2) in the form of an area. Specifically, this area delimits the region of the plot where the slowdown is kept within 10%, meeting the MCSs slowdown threshold. We compute this area as 10% slowdown to the 99th percentile of the *solo* configuration, normalized by its average value. Note that the actual region for this area may exceed 10%, as it is computed to account for both the 10% slowdown and the timing variability of the 99th percentile results (i.e., 99th percentile values normalized over the average). We evaluate the performance of the MBMSs by checking their ability to keep the MCS slowdown within this area.

MCS slowdown

Figures 5.5a, 5.5b, and 5.5c present the MCS slowdown results. Across all *duty_cycle* configurations, the plots show that when the MCSs are shorter than $20\mu s$ both MBMSs fail to mitigate the effect of memory interference, leading to a slowdown that exceeds δ_{10}. This result is caused by limits of the *reconfiguration techniques* used by the MBMSs: i) the **reconfiguration latency** - determined by the delay of the asynchronous IPI in *ASYNCHRONOUS* and the *reconfiguration period* in *SYNCHRONOUS* - which cause the MCSs to execute completely unprotected. ii) The **reconfiguration overhead** - caused by the system calls in *ASYNCHRONOUS* and the periodic checks in *SYNCHRONOUS* - which impacts negatively the duration of the MCSs. As a result, those MCSs fall below the control granularity of both *reconfiguration techniques* and are not protected by the MBMSs.

As MCSs grow in length, the MBMSs mitigate the slowdown in different ways: *ASYNCHRONOUS* enters δ_{10} when MCSs are longer than $\approx 100\mu s$, while *SYNCHRONOUS*, in its finest-grain configuration ($100\mu s$), enters δ_{10} when MCSs are longer than $\approx 1900\mu s$. Note that the *duty_cycle* parameter significantly impacts the performance of *SYNCHRONOUS*. When MCSs are sparse (Subplot 5.5a), the MBMS does not control memory interference for MCSs shorter than its *period*. On the other hand, as MCSs become more frequent (Subplot 5.5c), the performance of *SYNCHRONOUS* improves, and the MBMS controls memory interference for shorter MCSs as well. This occurs because the periodic checks performed by *SYNCHRONOUS* are more likely to identify MCSs when the latter are more frequent.

Overall, *ASYNCHRONOUS* shows 19x improved control granularity compared to *SYNCHRONOUS*, proving that different *reconfiguration techniques* may have a radical impact on the overall performance of a MBMS.

Bandwidth inside and outside MCSs

Figures 5.6a-5.6c and 5.6d-5.6f report the *bandwidth inside* and *outside* MCSs (definitions in Section 5.3) perceived by the *other* tasks. Specifically, the plots report the bandwidth results as a percentage, compared to the bandwidth

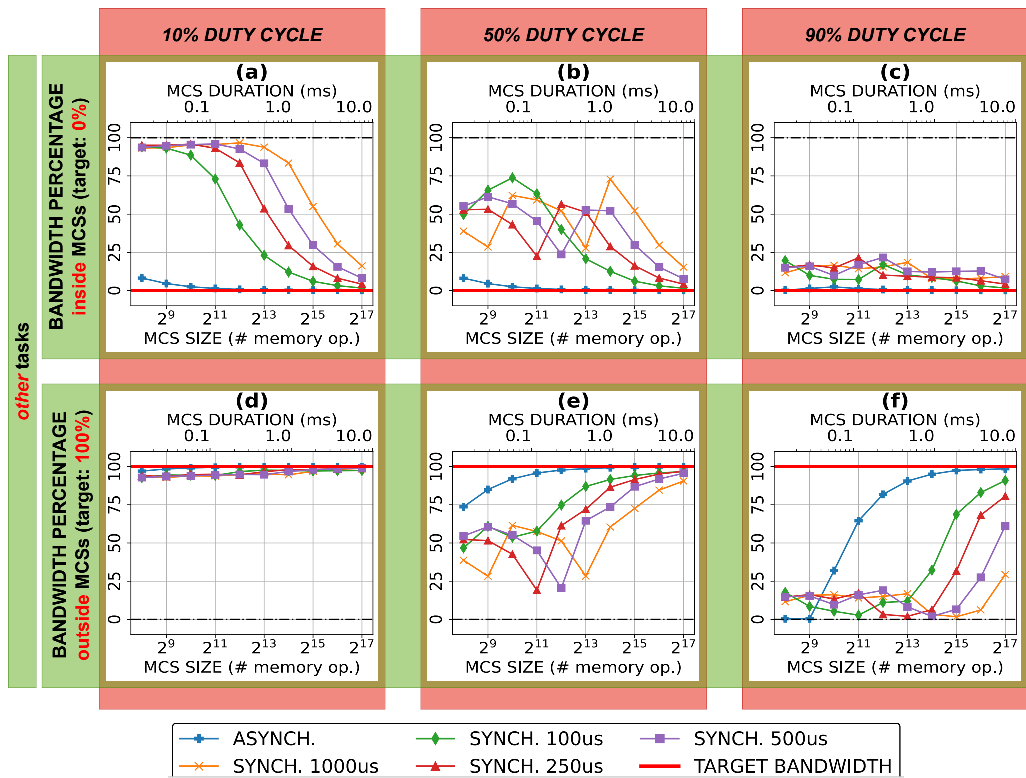


Figure 5.6: *BANDWIDTH PERCENTAGE* perceived by *other* tasks *inside* and *outside* the MCSs.

obtained in the *unregulated* running configuration. The expected values, denoted as target bandwidth in the plots, are 0% for *bandwidth inside* MCSs and 100% for *bandwidth outside* MCSs.

Bandwidth results are primarily influenced by the *duty_cycle* configuration of *main*. Figures 5.6a and 5.6d, – presented in the leftmost column – show the results obtained with the 10% *duty_cycle* configuration (i.e., low). In this configuration, the MBMSs allow *other* tasks to fully consume memory bandwidth *outside* MCSs (Figure 5.6d), while struggling to control the memory bandwidth consumed by *other* tasks *inside* MCSs (Figure 5.6a). This occurs because, in this configuration, the *main* task spends the majority of time outside MCSs, making it difficult for the MBMSs to detect the execution of MCSs and, in turn, stall the *other* tasks. Specifically, Figure 5.6a reports that *ASYNCHRONOUS* reaches the target bandwidth when MCSs are longer than $\approx 100\mu s$. Conversely, *SYNCHRONOUS* reaches the target bandwidth when MCSs are longer than $\approx 7ms$.

On the other hand, when the *duty_cycle* is high, the *main* task spends most of its time inside MCSs. As a consequence, the MBMSs may fail at promptly reactivating *other* tasks outside MCSs. This scenario, depicted in Figures 5.6c and 5.6f, reveals an aspect that is neglected by the MCS slowdown analysis performed in Section 5.5.3, i.e., the underutilization of the memory bandwidth outside MCSs. In fact, Figure 5.6c shows that both MBMSs correctly stall *other* tasks inside MCSs, while Figure 5.6f demonstrates that the MBMSs re-allocate full memory bandwidth to *other* tasks with different control granularities. *ASYNCHRONOUS* reaches the target memory bandwidth when MCSs are longer than $\approx 2ms$. This translates into $\approx 220\mu s$ interleaving time, consistent with the control granularity discussed in previous results. Vice versa, *SYNCHRONOUS* never reaches full memory bandwidth in our observation range.

Lastly, Figures 5.6b and 5.6d report the results for the 50% *duty_cycle* configuration. As expected, the plots report a combination of the two previous *duty_cycle* configurations. *ASYNCHRONOUS* reaches the target bandwidth *inside* and *outside* MCSs when both MCSs duration and interleaving time are longer than $\approx 200us$. *SYNCHRONOUS*, in the $100\mu s$ *period*

configuration, reaches the target bandwidth when both MCSs and interleaving times are longer than $\approx 7ms$.

The *bandwidth* results confirm the ones regarding MCS slowdown (see Section 5.5.3), demonstrating that the *asynchronous reconfiguration* technique improves the control granularity of the *synchronous* one not only in mitigating memory interference during MCSs, but also in re-allocating memory bandwidth outside MCSs.

5.6 Real-World Benchmarks Evaluation

This section presents the evaluation results obtained using the MBMSs to protect real-world workloads. This analysis enables us to verify the findings discussed in Section 5.5 on a real-world setup.

5.6.1 Evaluation Setup

In this evaluation, we leverage a benchmark that reflects the one used in the synthetic benchmark evaluation (see Section 5.5), substituting the *main* task with the tasks from the *PolyBench* benchmark suite [95]. Specifically, we execute the tasks from the *PolyBench* suite individually (i.e., one at a time) as the *main* task, co-scheduled with *other* tasks.

***COARSE*- and *FINE*-grain Regulation**

The benchmark can be executed in *solo*, *unregulated*, and *regulated* running configurations, varying the MBMSs in use (see Section 5.3). In the *regulated* running configuration, each *PolyBench* task can be configured to be protected by the MBMSs with two levels of granularity: *COARSE*-grain, where it is treated as task-long MCSs [121, 51], and *FINE*-grain, where only specific sections within its code are promoted to MCSs.

We use both *COARSE*- and *FINE*-grain approaches for multiple reasons. First, no systematic approach exists to identify MCSs in tasks' code. Therefore, protecting the entire benchmark (i.e., *COARSE*-grain) is the only

feasible starting point, which acts as a baseline to evaluate FINE-grain configurations. Second, the *COARSE*-grain approach may expose slowdown effects due to the overhead of the reconfiguration techniques of the MBMSs. Lastly, the *FINE*-grain approach allows us to analyze the granularity of the MBMSs in detecting MCSs.

The choice of MCSs when using the *FINE*-grain approach is based on static code analysis, identifying memory-intensive code regions within the *PolyBench* tasks' code. This selection aims to demonstrate the effectiveness of the MBMSs in mitigating memory interference, not to provide any timing guarantee for the execution of the entire tasks. The study of a systematic approach for identifying MCS candidates and ensuring tasks timing guarantees is beyond the scope of this analysis and is left for future work.

MBMSs Configuration

The MBMSs are configured as discussed in Section 5.4. However, to provide cleaner and more readable plots, we limit the *period* configurations of *SYNCHRONOUS* to the two that showed the best results in the previous evaluation: $100\mu s$, as the one with optimal control granularity, and $1000\mu s$, as the one with minimal overhead.

PolyBench Tasks Selection

In this analysis, we aim to evaluate a comprehensive range of MCS durations and interleaving times (short/long, sparse/frequent MCSs). To achieve this, we select three *PolyBench* tasks that – for our MCS configurations – provide a coverage of these different scenarios: `FDTD-2D`, `ADI`, and `GRAMSCHMIDT`.

Specifically, the results we report are composed of the *MCSs slowdown*, suffered by the *PolyBench* tasks, and the *bandwidth* observed both *inside* and *outside* the MCSs, enjoyed by the *other* tasks. Note that in the *COARSE*-grain configuration, the *MCS slowdown* corresponds to the overall slowdown of the entire *PolyBench* tasks, as they are treated as task-long MCSs in this setup.

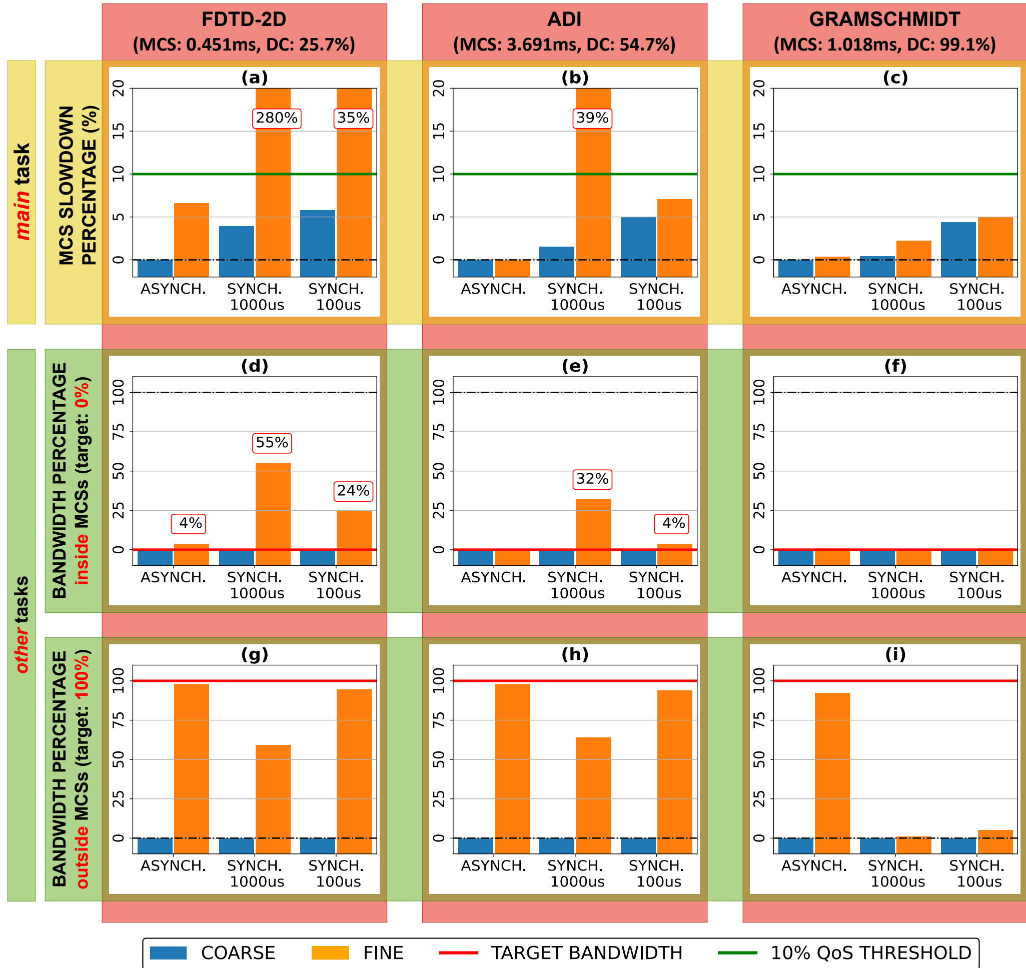


Figure 5.7: *MCSs SLOWDOWN* and *BANDWIDTH PERCENTAGE* inside and *outside* MCSs obtained using the *PolyBench* tasks.

5.6.2 Evaluation results

Figure 5.7 presents the results obtained by the three *PolyBench* tasks, one per column. Within the figure, the subplots report the *MCS slowdown* and the *bandwidth* results as a function of the level of protection (i.e., *COARSE* or *FINE*) and the MBMS in use (i.e., *ASYNCHRONOUS*, *SYNCHRONOUS* 100μs, *SYNCHRONOUS* 1000μs). Specifically, the *MCS slowdown* results are reported as the slowdown percentage of the *regulated* running configurations compared to the *solo* one. In the same way, the *bandwidth* results are reported as a bandwidth percentage compared to the one obtained in

unregulated running configuration.

COARSE*-grain *regulation

When using the *COARSE*-grain configuration, the MBMSs are active for the entire execution of the *PolyBench* tasks. This causes *other* tasks – which are stalled for the entire duration of the benchmark – to perceive 0% *bandwidth* both *inside* (Subplots 5.7d-5.7f) and *outside* (Subplots 5.7g-5.7i) MCSs. However, Subplots 5.7a-5.7c show that the *MCS slowdown* results do not correspond to 0%. In fact, for the two configurations of *SYNCHRONOUS*, the subplots show an additional slowdown, caused by the overhead (i.e., the *periodic checks*) of the *reconfiguration* technique. This corresponds to slowdowns up to 6% and 4% for the 100μs and 1000μs configurations of *SYNCHRONOUS*, which are consistent with the overhead results shown in Figure 5.4.

FINE*-grain *regulation

When using the *FINE*-grain configuration, `FDTD-2D`, `ADI`, and `GRAMSCHMIDT` generate different scenarios in terms of MCS durations and interleaving times. These scenarios depend on our selection of MCSs, i.e., the amount of code we promoted to MCSs within the tasks.

`FDTD-2D` generates a scenario where MCSs are short (i.e. $\approx 450\mu s$), and sparse (i.e., $\approx$ 25% *duty_cycle*). As discussed in Section 5.5.3, this scenario challenges the MBMSs to detect the execution of MCSs, and in turn, to stall *other* tasks during MCSs. Subplots 5.7a and 5.7d demonstrate this. Specifically, the subplots show that *ASYNCHRONOUS* controls the bandwidth consumed by *other* tasks *inside* MCSs down to 4%, ensuring *MCS slowdown* of 6% (i.e., within δ_{10}). On the other hand, the two configurations of *SYNCHRONOUS* only control the memory bandwidth down to 55% and 24%, causing the *MCSs slowdown* results to exceed δ_{10}. This results confirms the ones discussed in Section 5.5.

`GRAMSCHMIDT` generates a scenario where MCSs are longer (i.e., $\approx 1ms$), and frequent (i.e., $\approx$ 99% *duty_cycle*). In this case, the *PolyBench* task

spends most of its time executing MCSs. Therefore, the MBMSs effectively stall *other* tasks during MCSs, while struggling to re-activate them in the short MCS interleaving times. This, in turn, causes the underutilization of memory bandwidth *outside* MCSs. This scenario is depicted in Subplots 5.7c and 5.7f, where all the MBMSs correctly control the memory *bandwidth* perceived by *other* tasks *inside* MCSs down to 0%, reducing the *MCSs slowdown* within δ_{10}. Subplot 5.7g shows that, in the short MCS interleaving time (i.e., $\approx 10\mu s$), *ASYNCHRONOUS* enables *other* tasks to reach 92% of the available bandwidth, whereas *SYNCHRONOUS* reaches 5% in its best configuration. Regarding *ASYNCHRONOUS*, this outcome improves the one reported in Section 5.5, showing that the MBMS can achieve a control granularity of nearly $\approx 10\mu s$.

`ADI` generates a scenario where MCSs are long (i.e., $\approx 3.7ms$) and evenly sparse (i.e., $\approx$ 54% *duty_cycle*). Subplots 5.7b, 5.7e, and 5.7h show that this scenario is correctly handled by *ASYNCHRONOUS* and *SYNCHRONOUS* $100\mu s$, as they both reduce the *MCS slowdown* within δ_{10} and ensure more than 90% memory bandwidth *outside* MCSs. On the other hand, *SYNCHRONOUS* $1000\mu s$ does not control memory bandwidth both *inside* and *outside* MCSs, causing the *MCSs slowdown* to reach 39%. This is caused by the limited precision of the *synchronous reconfiguration* technique when configured with a $1000\mu s$ period. Ultimately, this result confirms the one discussed in Section 5.5.

5.7 Discussion

The goal of this work is to evaluate the performance impact of two basic *reconfiguration* techniques in a MBMS. To isolate the impact of such techniques, the only option is to turn off the *regulation* technique in the MBMSs, consequently nullifying its effect on the results. However, as discussed in Section 5.2, the performance of a MBMS (i.e., timing granularity and overall bandwidth utilization) is determined by the latencies and overheads of both the *reconfiguration* and the *regulation* techniques. In this section, we discuss how the additional latency and overhead of *regulation* techniques may impact

the performance of a MBMS.

In the MBMSs we discuss, the *reconfiguration* and *regulation* steps happen consecutively. First, the *reconfiguration* technique detects the start of a MCSs. Then, the *regulation* technique effectively enforces limited memory bandwidth to cores executing out-of-MCS tasks. As a result, the overall latency of the MBMSs is given by both the *reconfiguration* latency and the *regulation* latency. Specifically, given a *regulation* technique (e.g., based on bandwidth thresholds [124, 51]), the *regulation* latency adds up to the *reconfiguration* latency. Therefore, the latency results of the MBMSs we present in this work are independent of the *regulation* technique used, and can be considered as a baseline of the *reconfiguration* step of a MBMS.

While *reconfiguration* latency remains unaffected by *regulation*, the overhead is not: the choice of *regulation* technique can influence the overhead introduced by the *reconfiguration* techniques, and in turn, that of the MBMSs. The task model we adopt – based on a single task executing MCSs (see Section 5.1) – can be expanded to support multiple tasks executing MCSs simultaneously. This, for instance, would lead to the adoption of a *regulation* technique that enforces variable bandwidth thresholds to out-of-MCS tasks, computed based on the number and criticality level of the executed MCSs [3]. In such a scenario, *reconfiguration* points (i.e., tasks that enter/exit from MCSs) may occur concurrently across all cores in the system. This would impact the overhead generated by the two *reconfiguration* techniques in the MBMSs differently.

In *ASYNCHRONOUS*, each *reconfiguration* point requires sending an asynchronous IPI. Thus, the overhead of the MBMS increases with the number of MCSs that can be executed simultaneously. The overhead caused by multiple tasks triggering *reconfiguration* points resembles that generated by a single task executing short and frequent MCSs (low *MCS_size*, high *duty_cycle*). Thus, the plots reported in Figure 5.5 can provide an approximation of this scenario. On the other hand, in *SYNCHRONOUS*, the MBMS performs periodic checks at fixed time intervals, regardless of the number of *reconfiguration* points. Therefore, the overhead of the MBMS is independent of the number of tasks that execute MCSs.

This aspect entails a key difference between the two MBMSs: *ASYNCHRONOUS* ensures better control granularity when a single task executes MCSs, while *SYNCHRONOUS* may be better suited when the number of tasks executing MCSs increases. Ongoing work is currently focused on better exploring this latter scenario.

5.8 Conclusion

In this chapter, we provided a comprehensive analysis of the performance impact of two software-based *reconfiguration* techniques –*synchronous* and *asynchronous* – on a memory bandwidth management scheme (MBMS). The focus of the analysis was to compare how these *reconfiguration* techniques impact the ability of a MBMS to control a taskset with evolving bandwidth QoS requirements.

We implemented the two techniques in a MBMS based on *Memory-Critical Sections* (MCSs), which leverages them to detect when MCSs are executed in the system and accordingly activate or deactivate memory bandwidth *regulation*. To assess the performance of the two techniques, we conducted an evaluation on a real-world setup (i.e., a *Xilinx ZU9EG* platform) using a combination of synthetic and real-world benchmarks. Our evaluation focused not only on how precisely memory bandwidth *regulation* was enforced during MCSs, but also on how effectively bandwidth was re-allocated outside MCSs – an aspect often overlooked in memory bandwidth management. The results showed that the *asynchronous* technique significantly outperforms the *synchronous* one in terms of both control granularity and overall bandwidth utilization. In particular, the *asynchronous* technique improved the control granularity of the *synchronous* one by up to a factor of 19x, reducing granularity from the millisecond to the microsecond range.

Chapter 6

Mitigating GPU-to-CPU Memory Interference through Spatial GPU Throttling and Cuda Green Contexts

Heterogeneous systems-on-chip (HeSoCs) that integrate GPUs as hardware accelerators have become ubiquitous in embedded computing, enabling unprecedented computational throughput for data-parallel applications. However, hardware integration between CPU and GPU introduces a critical challenge: the shared memory subsystem becomes a fundamental bottleneck where CPU and GPU contend for limited bandwidth, creating unpredictable performance degradation due to bidirectional memory interference. GPU-to-CPU interference, in particular, is proven to be particularly severe [36], as the GPU massive memory bandwidth demand can significantly degrade the performance of tasks executing on the CPU cores. This problem is extensively studied in the context of mixed criticality systems [37, 104, 52], where the memory interference generated by memory intensive *best-effort* tasks running on the GPU can cause *critical* CPU tasks to miss their *deadlines*. For instance, Bechtel et al. show that, in a high-end vehicle, GPU-accelerated memory intensive tasks can cause *critical* CPU tasks to produce incorrect

results [24]. This has been so far the principal obstacle to the adoption of GPU-accelerated platforms in the context of mixed-criticality real-time applications.

In the literature, approaches capable of solving GPU-to-CPU memory interference operate primarily in the *temporal* domain. For instance, some approaches enforce time-synchronization between the CPU and the GPU, such as PREM-based approaches [52, 37], while others rely on exploiting periodic idleness of the GPU [90]. Although these approaches effectively mitigate GPU-to-CPU memory interference, they impose significant constraints. On the one hand, PREM-based approaches require the modification of CPU and/or GPU-accelerated tasks [52, 37]. On the other hand, the approaches exploiting periodic idleness of the GPU are limited by the coarse timing granularity at which GPU bandwidth regulation can be enforced [90]. These limitations severely limit their applicability in real-world systems.

Temporal-domain approaches have dominated the research world in this field, while other categories of approaches are comparatively underexplored. *Spatial*-domain regulation approaches offer an alternative strategy, where interference mitigation occurs by restricting the number of compute units available to GPU workloads. To the best of our knowledge, this category of has been limitedly investigated for GPU-to-CPU memory interference mitigation [68]. This gap – the absence of practical, implementable spatial-domain techniques for GPU bandwidth regulation on unmodified COTS hardware – motivates our investigation into the *spatial*-domain.

In this chapter, we address this gap by presenting a Memory Bandwidth Management Scheme (MBMS) that mitigates GPU-to-CPU memory interference by regulating GPU resource utilization along the *spatial* dimension. The MBMS operates on a simple principle: by reducing the number of Streaming Multiprocessors (SMs) available for GPU kernel execution (*partitioning*), GPU parallelism decreases, which in turn reduces GPU memory bandwidth consumption and diminishes memory interference.

We implement the MBMS on the two latest COTS HeSoCs provided by NVIDIA, the *NVIDIA AGX Orin* and the *NVIDIA AGX Thor*. Prior work implements NVIDIA GPU partitioning using features like NVIDIA *Multi*

Instance GPU (*MIG*) [97] or NVIDIA *Multi Process Service* (*MPS*) [103]. However, these features pose severe limitations in terms of both granularity and robustness of the generated GPU partitions [19]. In particular, NVIDIA *MIG* offers limited options in terms of GPU partition configurations [79]; while NVIDIA *MPS* allows multiple GPU tasks to concurrently execute on the GPU, but does not control which SMs each task is assigned to [19]. To overcome these limitations, in this analysis, we implement GPU partitioning using CUDA *Green Contexts*, another native feature provided by NVIDIA. CUDA *Green Contexts* enable the MBMS to be portable across all recent NVIDIA architectures, and provides fully software-driven, fine-grain, and dynamic control over the Streaming Multiprocessors on the GPU. We evaluate our spatial-domain MBMS on the *NVIDIA AGX Orin* and *NVIDIA AGX Thor* using both synthetic and real-world benchmarks, demonstrating that spatial SM regulation effectively reduces the memory interference induced by the GPUs in NVIDIA HeSoCs.

6.1 Background

In this section, we first recall the hardware and software model employed for this analysis. Following this, we provide an essential background on GPUs, with particular focus on the functional model, the internal architecture, and the programming model.

6.1.1 Hardware and Software Model

Figures 6.1 and 6.2 recall the architectural model of our reference hardware platforms: a *NVIDIA AGX Orin* and a *NVIDIA AGX Thor*. Both platforms are equipped with a multicore CPU and a GPU accelerator. In this model, GPU-to-CPU memory interference occurs at the system interconnect and the main memory level. We refer to this interference as *main memory interference.*

In this analysis, we focus on mitigating the memory interference generated by tasks executing on the GPU, as this interference can cause disproportional

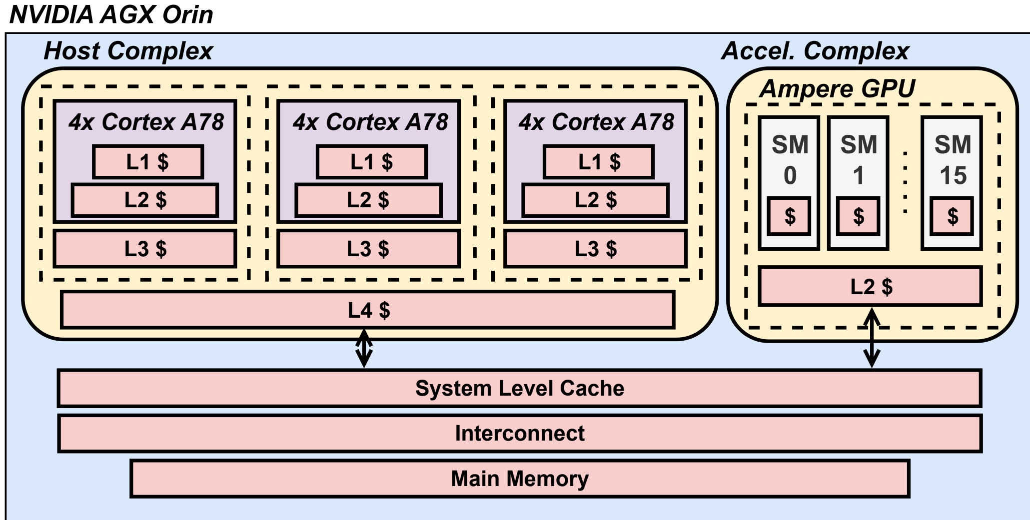


Figure 6.1: Architecture of *NVIDIA AGX Orin* development platform.

slowdown of tasks running on the CPU. To address this, we adopt a model in which CPU tasks can express *QoS requirements* as integer values, representing the maximum tolerated memory interference induced by the GPU. In particular, each CPU task specifies a QoS requirement in the range [0..100], where 0 allows unrestricted GPU execution and 100 requires GPU-induced interference to be minimal – which can be achieved by employing the most restrictive GPU regulation configuration possible. The global *QoS requirement* is computed as the maximum of all individual CPU task requirements. Based on this aggregated requirement, our proposed MBMS selects an appropriate bandwidth regulation level for the GPU.

Our evaluation employs both synthetic and real-world benchmarks. In both setups, the taskset comprises two CPU tasks: a *critical* task (the *main* task) executing time-critical workloads, and an *interfering* task (the *other* task) that executes GPU kernels generating memory interference. The *main* task represents the *time-critical* memory intensive task that must be protected by GPU-induced interference. Therefore, *main* can express *QoS requirements* of maximum tolerated GPU-interference, and its execution is protected using the MBMS.

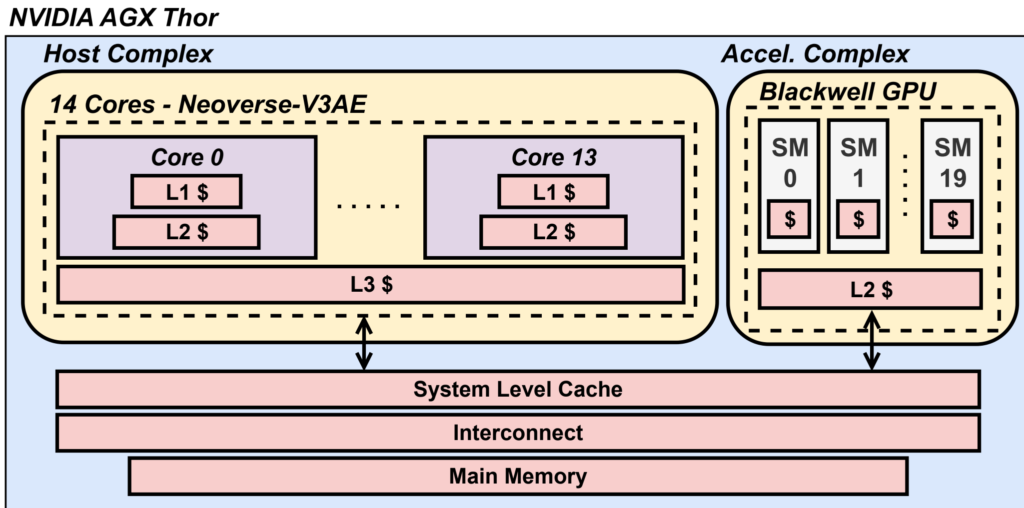


Figure 6.2: Architecture *NVIDIA AGX Thor* development platform.

6.1.2 Graphic Processing Units

Historically, Graphics Processing Units (GPUs) have been developed as specialized hardware accelerators limited to graphics rendering pipelines [1]. However, in the past decade, substantial advances in architectural flexibility and programmability have driven their adoption in diverse high-performance domains, ranging from scientific computing to machine learning and real-time systems. Today, General-Purpose Graphic Processing Units (GPGPUs) are designed to accelerate generic *data-parallel* workloads: workloads where the same processing logic must be applied across large set of independent data elements.

Functional Model

GPUs comply with the *Single Program Multiple Data* (SPMD) execution paradigm. In this model, a data-parallel computation is carried out by multiple threads in parallel: each thread executes the same program simultaneously on a distinct, private data element. The goal of the SPMD model is to maximize data-level parallelism. By mapping each thread to a distinct data element, this model enables GPUs to achieve massive concurrency and, in

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Algorithm 6: Generic Template of an SPMD Program

```
1 Function spmd_program(data):
      /* 1. Retrieve unique index                          */
2     tid ← GetGlobalThreadID()
      /* 2. Perform operations on private data             */
3     Operations(data[tid])
```

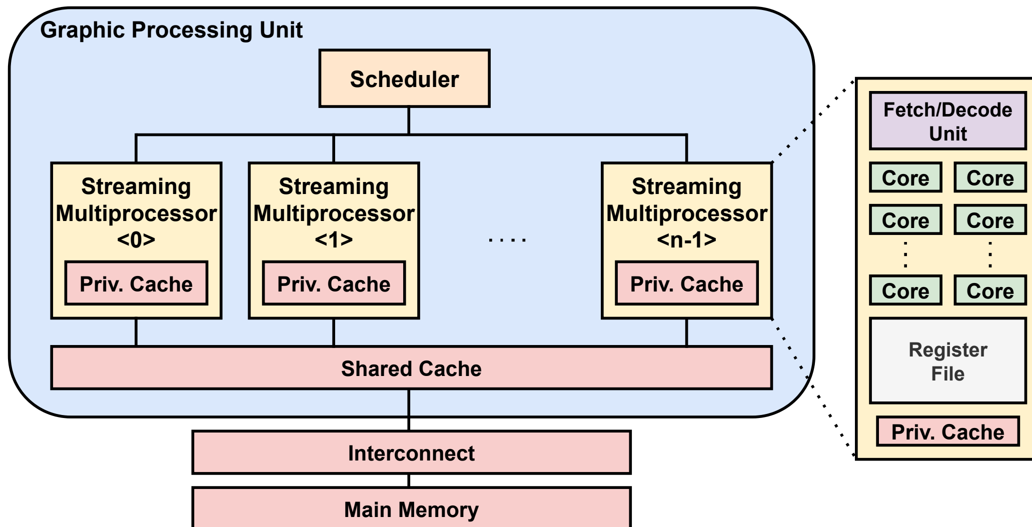


Figure 6.3: GPU architecture.

turn, high operational throughput.

Algorithm 6 presents the generic template of a SPMD program [44]. At runtime, this program is instantiated and executed by multiple threads. The execution begins with an *identity* phase: each thread retrieves its own global *identifier* (*tid*, Line 2). Then, each thread uses the *identifier* to perform operations on its own private data (Line 3).

GPU Architecture

To implement the SPMD paradigm efficiently, GPUs employ the specialized hardware architecture illustrated in Figure 6.3. At the top level, a GPU is organized as a scalable array of Streaming Multiprocessors (SMs), all connected to the shared memory hierarchy through multiple levels of cache memory.

Each SM functions as a multicore processor, equipped with a set of parallel execution cores and a massive *register file*.

At runtime, threads are allocated to a SM, where they persist for their entire execution. These threads – also named *active* threads – store their private architectural state (registers) in the partitioned *register file*. A key consequence of this design is that, unlike traditional CPUs, which swap contexts to memory, all *active* thread contexts remain resident on-chip. This way, the SM incurs zero overhead when switching between threads, achieving high operational throughput. The number of *active* threads an SM can support is strictly limited by the available hardware resources (e.g., registers) relative to per-thread requirements; however, this capacity is typically much larger than the number of physical cores. The ratio of *active* threads to the maximum hardware limit is defined as the *occupancy* level of a SM. Intuitively, higher *occupancy* means higher throughput, as the SM can execute more threads in parallel.

For execution, the SM manages threads in *warps*: groups of 32 threads that execute simultaneously on the parallel execution cores. A key architectural distinction from traditional multicore CPUs is that parallel cores in a SM share a single instruction *fetch/decode unit* (see Figure 6.3). Consequently, the execution of parallel cores proceeds in *lockstep*: in every clock cycle, the *fetch/decode unit* issues a single instruction that is executed simultaneously on all parallel cores. This way, *warps* of threads scheduled on parallel cores simultaneously execute the same instruction on different data. This hardware redundancy – multiple processing cores under a single *fetch/decode* unit – enables GPUs to achieve substantially higher throughput and lower energy consumption per operation compared to general-purpose CPUs.

GPU Memory Access Patterns

GPUs are hardware accelerators designed to execute highly data-parallel workloads that require substantial memory access bandwidth. In particular, GPUs can access memory both when GPU threads execute on SMs and

through DMA transfers. In this chapter, we focus on memory accesses performed by GPU threads on SMs.

When accessing main memory, each GPU thread individually issues a memory *request*, which is then served by main memory with one or more memory *transactions*. Threads typically retrieve data for computation based on their global identifier, which means that they also access memory locations according to this same identifier. In most workloads, this results in a common pattern: contiguous threads access contiguous memory locations [64]. NVIDIA optimizes this predictable pattern through memory access *coalescing* [2], where contiguous threads within a warp collectively issue a single memory *request* served by multiple back-to-back *transactions*. This optimization dramatically reduces the number of *requests* and, in turn, their overhead, leading to a significant improvement in memory bandwidth utilization and reducing access latency.

Coalescing breaks down when contiguous threads access non-contiguous (*strided*) memory locations. In such uncoalesced patterns, each thread must issue a separate memory request to retrieve its data, resulting in reduced bandwidth utilization and suboptimal memory access performance.

CUDA Programming Model

This thesis focuses on GPUs provided by NVIDIA, which can be programmed using the CUDA programming model. CUDA enables CPU tasks to *offload* data-parallel functions – called *kernels* – to the GPU for execution. CUDA *kernels* typically conform to the SPMD program template shown in Algorithm 6. The GPU instantiates and executes the *kernel* on a configurable number of parallel CUDA *threads*, each working on its own private data.

On the GPU, CUDA threads are organized following the hierarchical structure reported in Figure 6.4: at the first level, threads are grouped into *blocks*, which serve as the schedulable units on GPUs. At the second level, *blocks* are aggregated into a *grid*. When a *kernel* is executed, the corresponding *grid* of *blocks* is transferred to the GPU. The GPU scheduler then

[2] https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

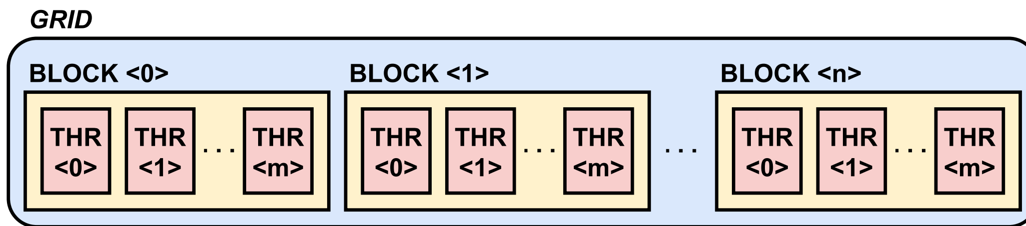


Figure 6.4: CUDA Threads hierarchical organization.

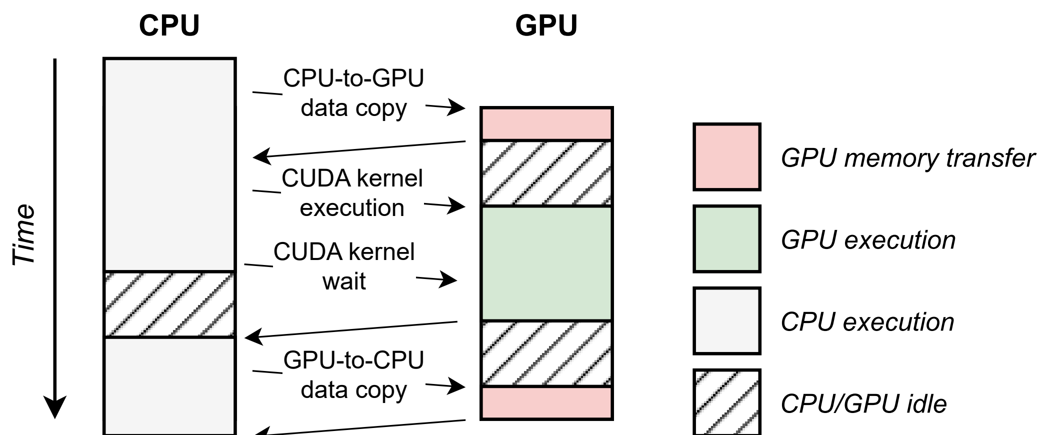


Figure 6.5: CUDA Offload Time diagram.

distributes *blocks* across available SMs. Once a *block* is assigned to an SM, the SM executes all its threads, scheduling multiple *warps* at a time (depending on the SM capability). Note that each SM can execute more *blocks* at the same time, while one *block* can only be mapped to a single SM. When all *blocks* finish execution, the *offload* finishes, and the CPU task can retrieve the result of the GPU computation.

In CUDA, the global *identifier* of each thread can be computed using two built-in variables: *blockIdx* and *threadIdx*. The *blockIdx* variable indicates the block containing the thread, while the the *threadIdx* variable contains the position of the thread within the block. Note that the values of these variables are private to each thread.

CUDA Offload Procedure

A typical CUDA GPU offload is composed of four main phases, shown in Figure 6.5. **i)** First, since the CPU and the GPU do not share the same virtual memory space, the data needed for the GPU computation are copied to GPU-mapped memory. **ii)** Second, the GPU kernel is executed. CUDA kernels can be executed within the application code as normal function calls, specifying the launch *grid* configuration through the *triple bracket notation*:

$$CUDA_kernel <<< num_blocks, threads_per_block >>> (...) \quad (6.1)$$

By executing this procedure, the CUDA *Runtime* – a software component handling NVIDIA GPUs – generates a kernel launch *grid* and executes the data-parallel kernel on the GPU. **iii)** Third, the CPU application waits for the GPU kernel result, which is available when the GPU *kernel* completes. **iv)** Finally, the result data from the GPU *kernel* execution are copied back to CPU-mapped memory.

These four phases are executed *asynchronously* by the CPU application, which means that the CPU application can continue execution while the GPU is performing memory transfers or computation. However, these phases are inherently sequential, as the following phase always depends on the previous one. Consequently, on the GPU side, computation and memory transfers cannot overlap, while the GPU hardware can in principle execute them concurrently. This leads to potential GPU resource underutilization. NVIDIA overcomes this limitation by introducing CUDA *streams*.

A CUDA *stream* is essentially an execution pipeline on the GPU. All commands issued in a *stream* (e.g., CPU-to-GPU copy, GPU execution, or GPU-to-CPU copy) are executed sequentially. However, multiple *streams* can be executed concurrently on a GPU. By distributing different CUDA *kernel* offloads across different *streams*, applications can overlap GPU memory transfers in one *stream* with GPU execution in another. This concurrency effectively improves the overall system throughput.

CUDA Green Contexts

When running on the GPU, every CUDA *kernel* is associated with a CUDA *context* – a virtual environment analogous to a CPU process that encapsulates its execution state, resource allocations, and device code [43]. CUDA *Green Contexts* (GCs) extend this abstraction, introducing spatial resource isolation. Unlike standard CUDA *contexts*, CUDA GCs represent lightweight execution contexts that are explicitly bound at creation time to a specific subset of GPU SMs [58]. Consequently, any CUDA *kernel* launched within a CUDA GC is strictly confined to execute only on its assigned SM partition.

In the existing literature, CUDA GCs are mainly employed to enable spatial multitasking, where multiple distinct CUDA *kernels* run concurrently on isolated partitions of the GPU to minimize interference [19]. In contrast, in this chapter, we exploit CUDA GCs for bandwidth regulation: by restricting a CUDA kernel to a subset of available SMs, we cap its maximum parallel request rate. This limits the aggregate memory bandwidth the GPU can consume and, in turn, the GPU-to-CPU memory interference.

CUDA *kernels* can be linked to CUDA GCs by exploiting CUDA *streams*. At initialization, a CUDA GC is associated with a CUDA *stream*. Consequently, any CUDA *kernel* launched into that *stream* is executed on the partition of SMs associated with the related CUDA GC.

6.2 *Spatial*-domain based MBMS

In this section, we detail the design, operation, and implementation of the MBMS based on *spatial*-domain regulation we propose.

6.2.1 Design

The idea behind *spatial*-domain regulation is that by limiting the amount of SMs available for GPU execution, the memory bandwidth consumed by the GPU decreases, and so does the GPU-to-CPU memory interference. The MBMS we propose instantiates this idea by intercepting GPU kernel launches

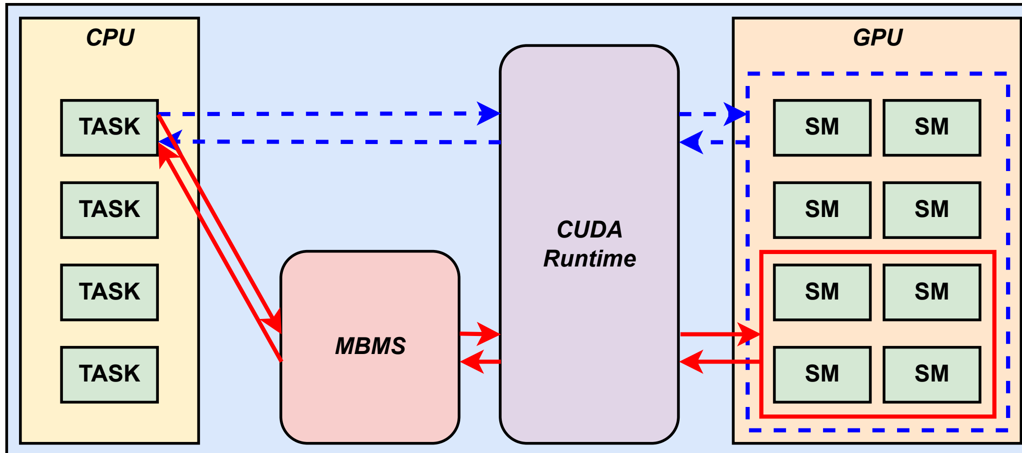


Figure 6.6: GPU *spatial*-domain regulation overview

and transparently modifying the launch configurations so that the kernels execute on a limited number of SMs.

Figure 6.6 presents a high level overview of GPU kernel launch process on a NVIDIA platform. As anticipated in Section 6.1.2, on NVIDIA platforms the GPU kernel launches are directly handled by the *CUDA Runtime*: the CPU application launches the *kernel* specifying the launch *grid*; the CUDA *runtime* instantiates that kernel on the GPU; and finally the CPU application retrieves the GPU kernel result. This operation is reported in the figure as a blue dashed line.

The MBMS we propose modifies this execution flow. Once loaded, the MBMS operates between CPU applications launching GPU kernels and the *CUDA Runtime*. This is reported in the figure as the red continuous line. In particular, when a GPU kernel is executed, the MBMS performs four fundamental steps: **i)** it intercepts the GPU kernel launch; **ii)** it checks the *QoS requirements* of the applications running on the CPU; **iii)** it modifies the GPU kernel launch configuration so that it executes on a partition of the GPU, according to the aggregated *QoS requirement*; and **iv)** it finally forwards the GPU kernel launch to the *CUDA Runtime*.

In both setups – i.e., when the MBMS is loaded or not – the CPU and the GPU execute concurrently, as CUDA kernel launches are performed *asynchronously* by the CPU applications.

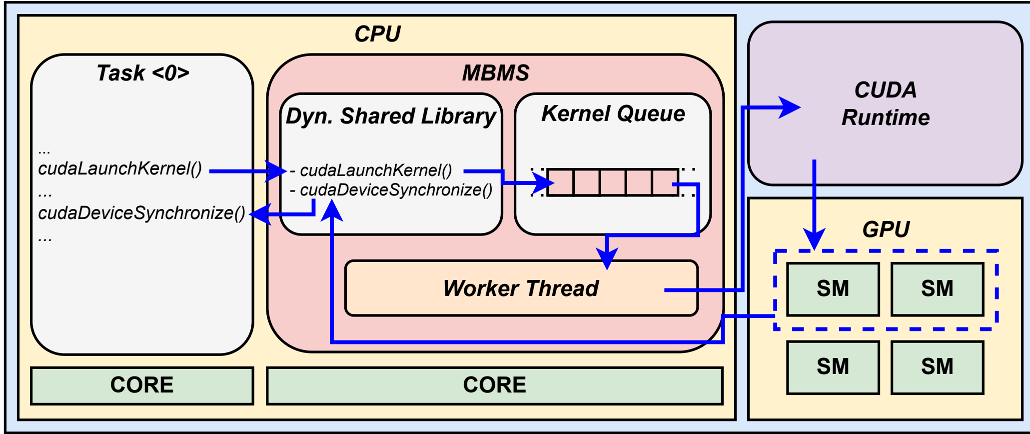


Figure 6.7: GPU *spatial*-domain regulation implementation.

6.2.2 Implementation

From the MBMS design description, four fundamental implementation components emerge: **i)** a mechanism to intercept GPU kernel launches; **ii)** a mechanism to monitor the *QoS requirements* of the applications running on the CPU; **iii)** a mechanism to modify GPU kernel launch configurations to enforce SM partitioning; and **iv)** a mechanism to forward kernels to the *CUDA Runtime*. We implement all these components by using specific strategies.

Figure 6.7 outlines the implementation of the MBMS, which comprises a dynamic shared library, a GPU kernel queue, and a *worker thread*. The dynamic shared library contains a redefinition of the *cudaLaunchKernel* function, originally exposed by the *CUDA runtime* to launch GPU kernels. At runtime, this library must be linked to GPU-accelerated applications. By using common library injection techniques – like the *LD_PRELOAD* mechanism on Linux distributions – we force the dynamic linker to load this library before standard system libraries. This enables the MBMS to intercept (*override*) the original *cudaLaunchKernel* function and, in turn, to intercept GPU kernel launches performed by the CPU applications. Upon intercepting the GPU kernel launch, the fake *cudaLaunchKernel* (i.e., the one provided by the library) enqueues the GPU kernel data structures into the dedicated queue, signals the *worker thread*, and immediately returns. This procedure *asynchronously* activates the *worker thread*.

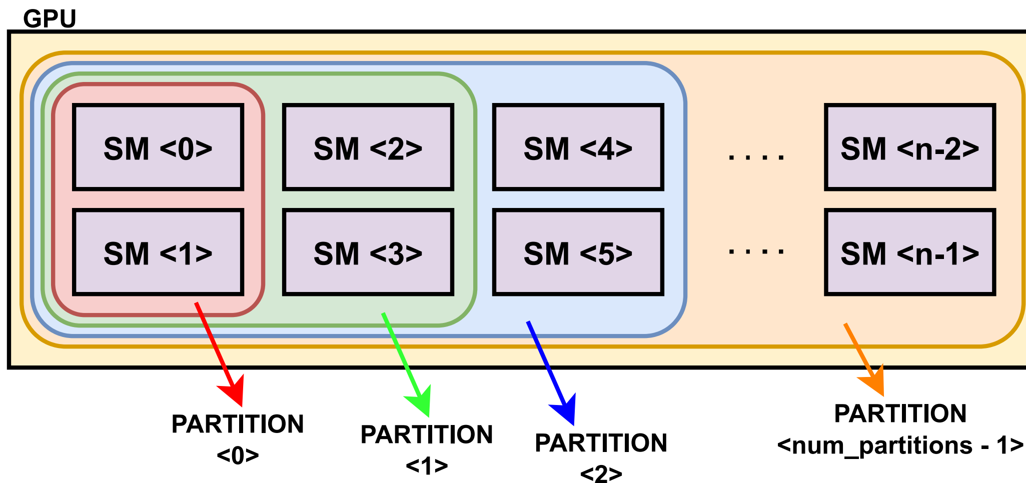
Algorithm 7: *Spatial*-domain based MBMS

```
1  num_partitions ← ...          /* number of GPU partitions */
2  kern_queue ← {}
3  Function worker_thread():
       /* CUDA GCs assigned to different GPU partitions    */
4      green_ctxs ← [0 ... num_partitions]
5      while True do
6          if kern_queue.empty() == True then
7              sleep()
8          end
           /* Wakes up here                                 */
9          kernel_task ← kern_queue.dequeue();
10         QoS_req ← max(GetCPUQoSRequirements());
11         green_ctx ← select_partition(QoS_req)
           /* executes kernel on CUDA GC                    */
12         real_cudaLaunchKernel(kernel_task, green_ctx);
13     end
```

Algorithm 7 reports the structure of the *worker thread*. At the initialization, this thread allocates a number of CUDA GCs (Line 4), each associated with a specific GPU SMs partition (see Section 6.1.2). When signaled, the *worker thread* performs four fundamental operations. First, it dequeues GPU kernel data structures previously enqueued by the fake *cudaLaunchKernel* (Line 9). Second, it retrieves the *QoS requirements* of all *critical* CPU applications (Line 10), which are stored in a dedicated memory region shared between CPU applications and the MBMS. Third, it selects an appropriate GPU SM partition based on the aggregated *QoS requirements* (Line 11), where smaller partition size means tighter regulation. Fourth, it executes the kernel on that SM partition. To map a CUDA kernel to a specific partition, the *worker thread* executes the original *cudaLaunchkernel* function in the CUDA GC yielding that partition (Line 12). This can be done using CUDA *streams* (see Section 6.1.2).

The dynamic shared library (see Figure 6.7) also implements a redefini-

Figure 6.8: GPU SM Partitioning



tion of the *cudaDeviceSynchronize* function, a *CUDA Runtime* function used by the CPU to wait for all kernels on the GPU side to finish execution. This function is typically executed to make sure that the GPU finishes computation before retrieving the results. In our MBMS, this condition occurs when all GPU kernels finish their execution on the GPU, and the kernel queue is empty.

GPU SM Partitioning

Figure 6.8 sketches how partitioning happens on the GPU. Specifically, a GPU has multiple partitions, each of increasing size. The smallest partition (i.e., the one that enforces maximum regulation) is composed by two SMs – a limitation imposed by CUDA GCs [58]. Partitions then grow two SMs at a time (4,6,8 SMs...), allowing the GPU to implement more parallelism, and in turn, to consume more memory bandwidth. Intuitively, for a GPU, the number of partitions is half the number of the SMs. The mapping between the aggregated *QoS requirement* level of the CPU – in the range [0..100] – and the SM partition to be used on the GPU is performed by the *worker thread*. Specifically, this mapping depends on the possible number of SM partitions on the GPU: chunks of *QoS requirement* levels are mapped to

each SM partition. For instance, on a 20 SM GPU (with 10 partitions), chunks of 10 *QoS requirement* levels are mapped to each SM partition: *QoS requirement* in range [0..10] are mapped to the largest partition (20 SMs), while QoS requirements level in range [90..100] are mapped to the smallest partition (2 SMs). In this chapter, we do not further discuss this mapping, as in the evaluation we always study all possible SM partitioning configurations, regardless of the QoS requirement levels of the CPU.

6.3 Synthetic Benchmarking

This section presents the synthetic benchmarks and memory traffic generators we use in the evaluation. Specifically, we employ a synthetic benchmark on the CPU side, representing the time-critical workload to be protected from GPU-induced interference, and two memory access generators on the GPU side, causing memory interference. In the following, we discuss the details of the three workloads.

6.3.1 *cpu_synth*

cpu_synth is a memory-intensive synthetic benchmark directly derived from those seen in previous analyses (See Section 5.3). Its structure, presented in Algorithm 8, is based on a memory-intensive loop (Lines 2-7). At runtime, during each iteration of the loop, the benchmark times a chunk of memory operations (Line 4), and stores the result (Line 6). When the last iteration ends, *cpu_synth* returns a list containing all the iteration durations (Line 8).

As seen in previous analyses (see Section 5.3), this memory-intensive benchmark enables us to directly analyze the level of memory interference in the system. In fact, by timing the memory traffic generated by the benchmark and comparing the results under different interference configurations, we evaluate the level of memory interference in the system.

We configure *cpu_synth* to access a memory buffer larger than the system level cache ($fp = 64MB$) with a read-only strided access pattern. In particular, the benchmark performs load operations (`ldr`) separated by a *stride*

Algorithm 8: CPU Synthetic Benchmark with Timing

```
1  Function cpusynth(stride):
       /* Number of memory-intensive loop iterations          */
       Data: iterations
       /* List of memory operations durations                 */
       Data: execution_times [0 ... iterations]
2      for idx ← 0 to iterations do
3          iter_start ← time();
           /* Perform a chunk of memory operations            */
4          memory_ops_chunk(stride);
5          iter_stop ← time();
6          execution_times[idx] ← iter_stop − iter_start;
7      end
8      return execution_times;
```

of 64B, as this value corresponds to a cache line size on our reference platforms. As in previous analyses (see Section 5.5.2), the choice of this memory access pattern is motivated by several aspects: i) Thanks to the memory buffer size and the stride, each memory load operation causes main memory access. This makes this benchmark particularly sensitive to memory interference. ii) Since the memory traffic is read-only, the cache pollution (i.e., dirty cache lines) is limited, and the number of spurious cache writebacks – which would cause results instability – is minimized. For statistical significance, we configure *cpu_synth* to perform 1000 iterations.

6.3.2 GPU Memory Traffic Generators

This analysis aims at evaluating the effectiveness of our proposed *spatial*-domain based MBMS in controlling GPU-to-CPU memory interference. To generate interference, we employ synthetic memory traffic generators on the GPU, as they enable systematic control over the GPU memory traffic. By using such generators, we can evaluate different memory traffic types (*read*, *write*, and *readwrite*) and different memory access patterns (*coalesced* and

Algorithm 9: *spmd_traffic_generator* Kernels & Host Structure

```
1  threads_per_block ← 128 ;                    /* Fixed block size */
2  Kernel gpu_kern(trf, stride):
       /* Retrieve global thread idx                             */
3      identifier ← blockIdx × blockDim + threadIdx
4      mem_offset ← identifier × stride;
       /* Perform memory operation                               */
5      mem_access(trf, mem_offset);
6  Function cpu_main(num_ops, trf, stride):
7      nblocks ← num_ops/threads_per_block;
8      for idx ← 0 to ∞ do
9          gpu_kern
             <<< nblocks, threads_per_block >>> (trf, stride);
10     end
```

strided, see Section 6.1.2). By testing these diverse configurations, we comprehensively assess the effectiveness of the MBMS across a wide range of GPU memory interference scenarios.

spmd_traffic_generator

spmd_traffic_generator is a GPU memory traffic generator inspired by the *HeSoC-mark* benchmark suite [94]. Its structure, shown in Algorithm 9, is based on a CPU task (Line 6) that iteratively launches memory intensive GPU kernels (Line 2). The GPU kernels generate configurable main memory traffic by accessing memory with configurable traffic type and stride. As a result, this benchmark enables us to generate configurable GPU memory interference on our reference platforms.

The structure of the GPU kernel executed by *spmd_traffic_generator* (Line 2) adheres to the SPMD template shown in Algorithm 6. Therefore, the GPU kernel is instanced on a number of GPU threads, and each thread performs the same memory operation at a different memory location. The type of memory operation performed by the threads and the memory loca-

tions accessed are defined by the parameters of the GPU kernel: *trf* and *stride* (Line 2). Specifically, the *trf* parameter regulates the memory traffic type, which can be *read*, *write*, or *readwrite*. These traffic types execute loads, stores, or load and stores operations, respectively. The *stride* parameter regulates the memory access pattern, defining the distance in bytes between the memory locations accessed by contiguous GPU threads.

The execution of *spmd_traffic_generator* begins with the CPU task. Initially, the CPU task configures the GPU kernel launch *grid*, determining the number of GPU threads that execute the kernel. In this generator, each GPU thread performs a single memory operation. Therefore, the number of threads in the *grid* must correspond to the number of memory operations required per GPU kernel. The number of memory operations per GPU kernel is given as input to the generator – indicated as *num_ops* in the algorithm. *spmd_traffic_generator* employs a fixed block size of 128 threads (Line 1), as we empirically find this value to be the one that maximizes SM *occupancy* (see Section 6.1.2) in our evaluation. Given the value of *num_ops* and the 128 threads per block, the number of blocks in the launch *grid* to be executed can be computed as $nblocks = num_ops/128$ (Line 7).

After configuring the launch *grid*, the CPU task iteratively launches the GPU kernel. According to the SPMD model, the execution of the GPU kernel starts with an *identity* phase (Line 3). In this phase, each thread retrieves its own global *identifier* using the *blockIdx* and *threadIdx* variables (see Section 6.1.2). The resulting *identifier* is then multiplied by the *stride* parameter (Line 4). Therefore, consecutive threads (i.e., threads with consecutive coordinates) access memory locations that are exactly *stride* bytes apart. Then, the execution of the GPU kernel continues by performing a single memory operation (Line 5), which targets the memory location computed using the retrieved *identifier*.

In the evaluation, we control the execution of *spmd_traffic_generator* using our proposed MBMS. In particular, we evaluate the interference it generates when scheduled on *partitions* of the SMs available on the GPUs. As stated in Section 6.1.2, the smallest SM partition that can be implemented using CUDA GCs is made up of two SMs. This may possibly mask the

full potential of *spatial* regulation, as the smallest partition can ideally be composed of only one SM. To evaluate the memory interference effectively generated by a single SM, we propose an additional GPU memory traffic generator named *spmd_traffic_generator_conf_sm*. This generator actively executes GPU kernels on a subset of the SMs available on the GPU, also reaching the one SM configuration.

spmd_traffic_generator_conf_sm

spmd_traffic_generator_conf_sm is a GPU memory traffic generator that builds on *spmd_traffic_generator*. Unlike its predecessor, which executes memory-intensive GPU kernels using the entire GPU, *spmd_traffic_generator_conf_sm* is designed to execute the kernels using a configurable subset of the SMs available on the GPU. This capability enables it to simulate the *spatial* regulation effects enforced by our MBMS, with the granularity of a single SM.

In order to execute GPU kernels using a configurable subset of the SMs, *spmd_traffic_generator_conf_sm* necessitates a mechanism to implement GPU kernel-to-SM affinity. However, NVIDIA does not provide any mechanism to implement affinity at the level of the single SM. Therefore, the generator simulates this behavior by employing a specific strategy: when launching GPU kernels, it allows only GPU threads that spawn on the desired set of SMs to execute. The threads that spawn on the other SMs are discarded. This way, since GPU threads do not migrate (see Section 6.1.2), the GPU executes kernels using only the desired SMs.

The choice of allowing only GPU threads that spawn on the desired SMs to execute breaks the standard SPMD template, as only a subset of the total GPU threads effectively execute the GPU kernel. In this scenario, the surviving GPU threads execute a portion of the full data-parallel workload iterations. To restore the execution of the full GPU kernel on a subset of the GPU SMs, *spmd_traffic_generator_conf_sm* implements GPU kernel that forces each thread to execute multiple iterations. Algorithm 10 presents this solution in detail.

Algorithm 10: *spmd_traffic_generator_conf_sm* Kernels & Host Structure

```
1  max_threads_per_sm ← 1536
2  threads_per_block ← 128
3  total_mem_ops ← 0
4  Kernel gpu_kern(num_ops, trf, stride, num_sm):
5  |  if get_smid() < num_sm then
6  |  |  virt_blockIdx ← get_virt_blockIdx();
7  |  |  mem_offset ←
   |  |    (virt_blockIdx × blockDim + threadIdx) × stride;
8  |  |  step ← stride × max_threads_per_sm × num_sm;
9  |  |  while total_mem_ops ≤ num_ops do
10 |  |  |  total_mem_ops ← total_mem_ops + 1;
11 |  |  |  mem_access(trf, mem_offset);
12 |  |  |  mem_offset ← mem_offset + step;
13 |  |  end
14 |  end
15 Function cpu_main(num_ops, trf, stride, num_sm):
16 |  blocks_per_sm ← max_threads_per_sm/threads_per_block;
17 |  sms_in_platform ← get_number_of_sms() ;
18 |  nblocks ← blocks_per_sm × sms_in_platform;
19 |  for idx ← 0 to ∞ do
20 |  |  total_mem_ops ← 0 ;
21 |  |  gpu_kern <<< nblocks, threads_per_block >>>
   |  |    (num_ops, trf, stride, num_sm);
22 |  end
```

The execution of *spmd_traffic_generator_conf_sm* begins with the CPU task configuring the GPU kernel launch *grid* (Line 15). Unlike the SPMD template, in *spmd_traffic_generator_conf_sm* each GPU thread executes multiple iterations of the data-parallel workload. For this reason, the number of threads in the launch *grid* does not depend on the number of memory operations that must be performed by the GPU kernel. Instead, in this generator, the launch *grid* configuration is statically designed to saturate the SMs (i.e., to maximize the *occupancy level*). This way, we can effectively evaluate the maximum memory interference generated by a subset of the GPU SMs. Both reference platforms (*NVIDIA AGX Orin* and the *NVIDIA AGX Thor*) support a maximum of 1536 *active* threads per SM[3]. As for *spmd_traffic_generator*, this generator uses fixed-size blocks of 128 threads (Line 2). Consequently, the launch *grid* is configured with 12 blocks of 128 threads per SM ($12 \times 128 = 1536$, Lines 17-18).

After configuring the launch *grid*, the CPU task iteratively launches the GPU kernel (Line 21). The execution of GPU threads – all executing the kernel (Line 4) – starts by retrieving the identifier of the SM where they spawned (Line 5). Based on this value, each thread selectively continues to execute only if its SM is one of the desired SMs. This occurs if its SM identifier is smaller than *num_sm* – a parameter of the GPU kernel. Since the mapping between GPU threads and SMs is not known in advance, it is impossible to predict the *identifiers* of the threads that survive the selection process. Consequently, it is also impossible to use the thread *identifiers* to determine the memory access pattern (like the SPMD template, see Algorithm 6). In this generator, GPU threads access memory using a virtual (i.e., custom) *identifier*. In particular, thread blocks that survive the SM selection process are assigned a virtual *identifier* (i.e., virtual *blockIdx*), which is unique for each block and ranges continuously from 0 to the number of thread blocks that survived. The threads in the survived blocks use this virtual *blockIdx* to compute their virtual *identifiers* (Line 7), and consequently access memory (Line 11). As in *spmd_traffic_generator*, the memory traffic type and access

[3]https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications-technical-specifications-per-compute-capability

patterns are regulated by the benchmark parameters *trf* and *stride*.

Each GPU thread performs multiple memory operations, until the total number of memory operations required by the kernel is completed (Line 9). Specifically, at each memory operation, the GPU threads increase the location of the memory access (i.e., the address) by the value of the variable *step* (Line 12). The value of this variable is computed as the number of threads that survive the SM selection process multiplied by the *stride* (Line 8). Therefore, each each thread iteratively accesses previously-unaccessed memory in increasing order, simulating the memory access pattern of multiple consecutive thread blocks being scheduled on the SMs of the GPU. This memory access pattern is consistent with the one triggered by *spmd_traffic_generator*.

6.4 Synthetic Benchmark Evaluation

In this section, we provide an extensive evaluation of the *spatial*-domain based MBMS we propose. In the following, we discuss the evaluation setup, the evaluation process, and the results.

6.4.1 Evaluation Setup

We perform the analysis on two state-of-the-art NVIDIA heterogeneous platforms: the *NVIDIA AGX Orin* and the *NVIDIA AGX Thor*. The hardware configuration of the two platforms is reported in Section 2.3.

From a software point of view, both platforms are equipped with the newest available *Jetson-Linux* distribution, a Linux distribution provided by NVIDIA. Specifically, the *NVIDIA AGX Orin* is equipped with *Jetson-Linux* 36.4.4 (Linux kernel version 5.15), while the *NVIDIA AGX Thor* is equipped with *Jetson-Linux* 38.2 (Linux kernel version 6.8.12). On both platform, we disable CPU, GPU, and main memory frequency scaling for the evaluation. Moreover, we disable the Graphic User Interface (GUI). This enables more stable and repeatable results.

6.4.2 GPU Memory Traffic Generators Configurations

We conduct the experiments using the *cpu_synth* synthetic benchmark running as the *main* task on the CPU and the GPU memory traffic generators as *interfering task*. In the following, we describe the configurations of the two GPU memory traffic generators (i.e., *spmd_traffic_generator* and *spmd_traffic_generator_conf_sm*).

spmd_traffic_generator* and *spmd_traffic_generator_conf_sm

We configure the GPU kernels of both GPU memory traffic generators to access memory using 16 bytes memory operations (i.e., 16 bytes loads, 16 bytes stores, or 16 bytes load + 16 bytes stores). Specifically, we do this by implementing `ld.volatile.global.v2.u64` and `st.volatile.global.v2.u64` operations in the GPU kernels. We select these operations because they are the largest that can be performed on these architectures. Therefore, they potentially maximize memory bandwidth consumption.

The two benchmarks share key parameters that deeply affect the GPU memory access pattern: *trf* and *stride*. In the evaluation, we configure these parameters to provide an extensive overview of different GPU memory access patterns. We configure the GPU to access memory with all supported memory traffics: *trf=read*, *trf=write*, *trf=readwrite*. Moreover, we use two *stride* configurations: 16B and 512B. Given our choice of using 16 bytes memory operations, these stride configurations correspond to two very different memory access patterns. When using 16B stride, the memory access pattern is contiguous, and therefore is maximally *coalesced* by the GPU (see Section 6.1.2). This means that only one thread for each *warp* issues a memory request, which is then served by the main memory with multiple back-to-back transactions. On the other hand, when using the 512B *stride*, the stride corresponds to the maximal single memory request width: a single memory request can move at most 16B data for 32 threads in a *warp* ($16B \times 32 = 512B$). Therefore, every GPU thread is forced to issue a different main memory request to retrieve its data. This configuration represents maximally *uncoalesced* memory traffic.

6.4.3 Evaluation Process

We repeat the following evaluation process on both our reference platforms.

First, we execute *cpu_synth* in isolation on the CPU, configured as described in Section 6.3. The result obtained with this execution serves as a baseline of the CPU memory access latency. Following that, we execute the GPU memory traffic generators in isolation (i.e., one at a time) with the goal of evaluating the memory bandwidth they consume when executed on a *partition* of the GPU. Specifically, we execute the generators with all the possible parameter configurations reported in Section 6.3: *trf*={*read*,*write*,*readwrite*} and *stride*={16,512}. For each possible combination of these parameters, we control the portion of GPU the generator can use in different ways, depending on the generator in use. When using *spmd_traffic_generator*, we control the partition of the GPU to be used through the *spatial*-domain based MBMS. On the other hand, for *spmd_traffic_generator_conf_sm*, we use the built-in parameter *num_sm*. Note that, when using *spmd_traffic_generator_conf_sm*, the generator enables us to evaluate GPU partitions containing as few as 1 SM, while the spatial-domain based MBMS only allows 2 SMs partitions.

Lastly, we co-schedule *cpu_synth* and the two GPU memory traffic generators (one at a time). The results obtained with this running configurations enable us to evaluate the slowdown induced on *cpu_synth* by the GPU memory interference. Specifically, we leverage all the aforementioned parameter configurations of *spmd_traffic_generator* and *spmd_traffic_generator_conf_sm*. Also in this case, we control the GPU SM allocation using the *spatial*-domain based MBMS for *spmd_traffic_generator* and the *num_sm* parameter for *spmd_traffic_generator_conf_sm*.

6.4.4 Evaluation Results

Our analysis evaluates the bandwidth consumption and slowdown caused by GPU kernels when executed on *partitions* of the GPU. The primary variables influencing these results are the GPU kernel parameters *trf* and *stride*. Therefore, we group the results based on these two parameters.

Figures 6.9, 6.10, 6.11, and 6.12 present the synthetic benchmark evaluation results. Each figure corresponds to a specific *stride* configuration: Figures 6.9 and 6.10 use *stride*=16, while Figures 6.11 and 6.12 use *stride*=512. Within each figure, each pair of subplots corresponds to a specific *trf* configuration (i.e., *read*, *write*, *readwrite*).

The pairs of subplots illustrate how bandwidth or slowdown (reported on the Y axis) vary as a function of the number of SMs utilized by the GPU (reported on the X axis) for a specific configuration of *trf* and *stride*. Low SMs values (the leftmost ones on the X axis) correspond to configurations where the GPU is maximally regulated, while high SMs values (the rightmost ones) correspond to the unregulated GPU configurations. In particular, each subplot contains the results obtained by spatially-limiting the GPU with the two approaches we discuss: **i)** the *CUDA Green Contexts*, and **ii)** the SM selection implemented by *spmd_traffic_generator_conf_sm*. Since the latter approach ideally corresponds to an enhanced version of CUDA GCs that allow 1 SM configuration granularity, we refer to it as *"CUDA Green Contexts 1 SM"* in the results.

As a slowdown metric, we report the average duration of the *cpu_synth* iterations under interference, normalized by the average duration in the solo configuration. We use averaged results because the standard deviation remains low (< 2%) relative to the mean. We measure GPU memory bandwidth consumption using a tool provided by NVIDIA, the *Central Activity Monitor* (*actmon*). The *actmon* monitors the fraction of active cycles at the memory controller level over an observation period. By multiplying this fraction by the nominal main memory bandwidth, we derive the actual bandwidth consumption over the observation period.

Evaluation Results, *stride*=16B (contiguous, coalesced)

Figures 6.9 and 6.10 present the results obtained by configuring the GPU kernels with *stride*=16B on the *NVIDIA AGX Orin* and the *NVIDIA AGX Thor*, respectively. This configuration generates contiguous memory accesses from the GPU threads. Therefore, the GPU can *coalesce* all the memory

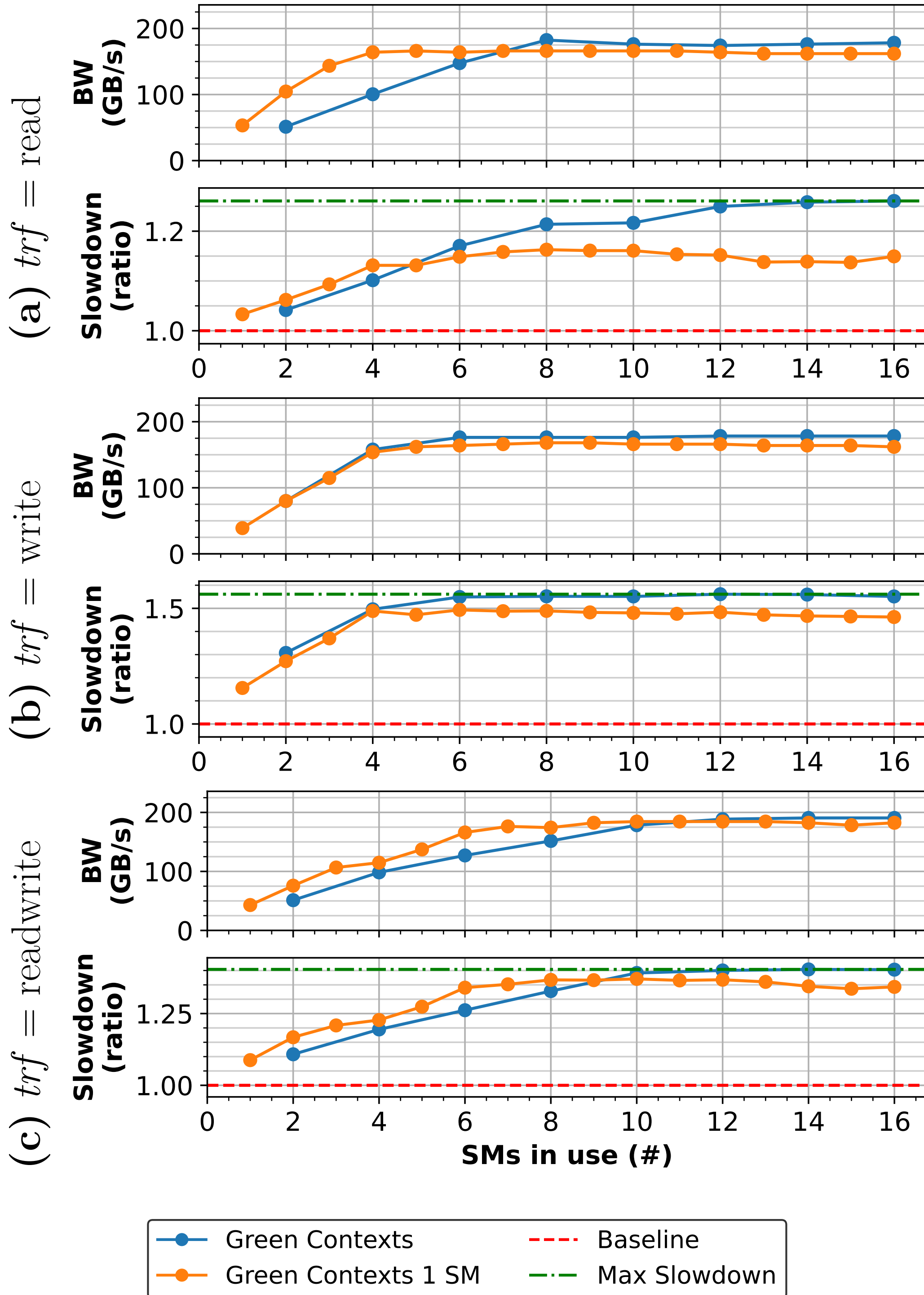


Figure 6.9: Synthetic benchmark results on *NVIDIA AGX Orin*, *coalesced*

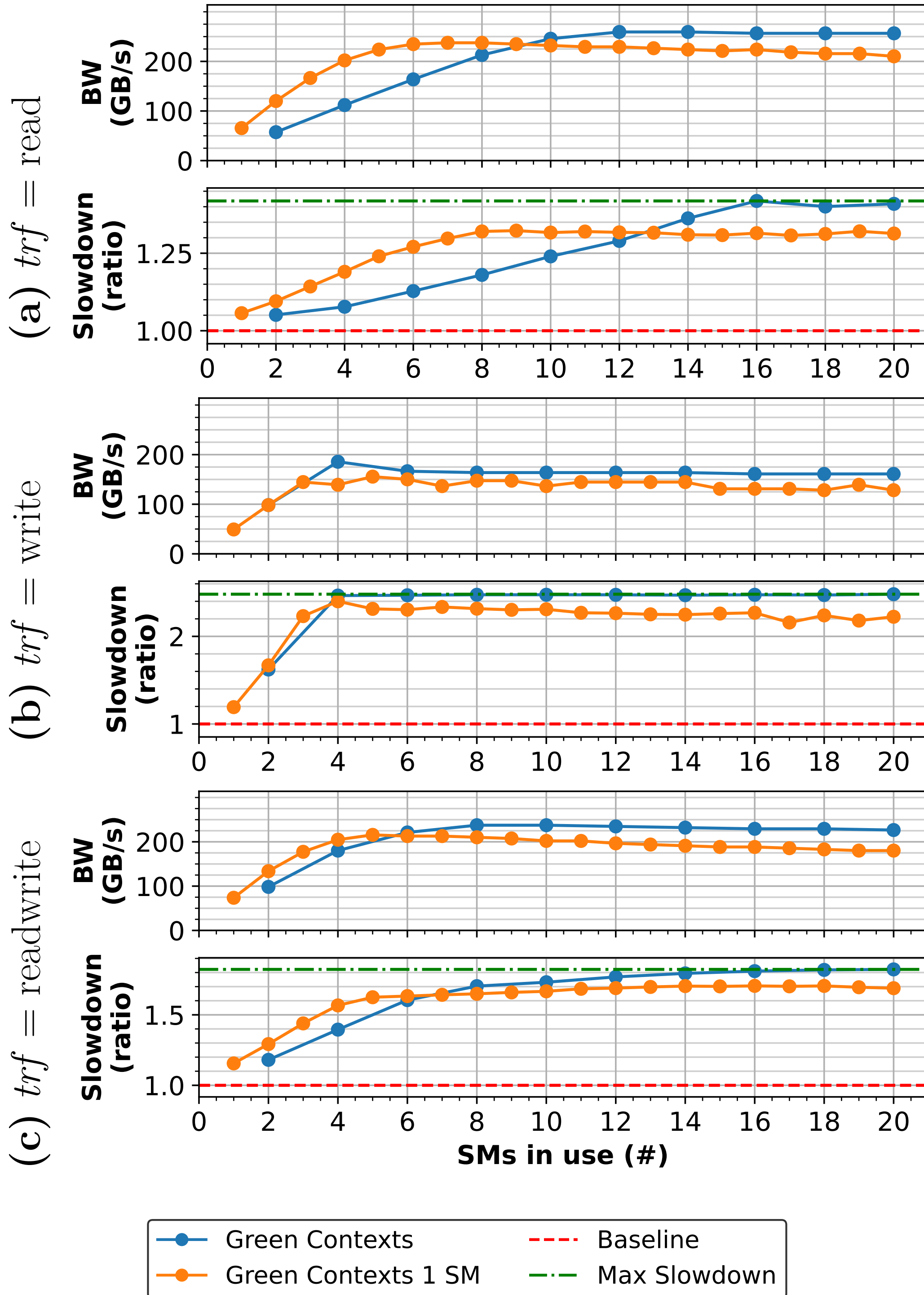


Figure 6.10: Synthetic benchmark results on *NVIDIA AGX Thor*, *coalesced*

requests performed by GPU threads into a single memory transaction per *warp*. This represents a highly optimized memory traffic for the GPU, and it enables very-high memory bandwidth consumption.

This aspect is confirmed in the figures, as this type of memory access pattern nearly saturates the physically available bandwidth on both platforms: 205GB/s and 273 GB/s for the *NVIDIA AGX Orin* and the *NVIDIA AGX Thor*, respectively. Specifically, on *NVIDIA AGX Orin*, the traffic that generates the highest bandwidth consumption is *trf=readwrite* (see Figure 6.9c), reaching 192GB/s of memory bandwidth (93.6% of the nominal bandwidth). On the other hand, on the *NVIDIA AGX Thor* the highest bandwidth memory traffic is *trf=read* (see Figure 6.10a), reaching 257GB/s memory bandwidth (94.1% of the nominal bandwidth).

On both platforms, the memory traffic that generates the highest slowdown is represented by *trf=write*, causing 1.55× slowdown of the *cpu_synth* task on the *NVIDIA AGX Orin* (see Figure 6.9b), and a more impressive 2.5× slowdown on the *NVIDIA AGX Thor* (see Figure 6.10b).

The figures show that, on both platforms, the *spatial*-domain MBMS regulates the memory bandwidth of the GPU memory traffic generator, in turn reducing the memory interference on the *cpu_synth*. Specifically, by gradually limiting the number of SMs used by the GPU-accelerated task, the MBMS reduces the slowdown caused by the *trf=write* GPU memory traffic from 1.55× to 1.2× on the *NVIDIA AGX Orin*, and from 2.5× to 1.6× on the *NVIDIA AGX Thor*. This proves the effectiveness of the MBMS, as the *coalesced* memory access pattern is most common one when the GPU is used in real-world scenarios [42, 69].

The slowdown results obtained by regulating the *num_sm* parameter of *spmd_traffic_generator_conf_sm* – reported as "*CUDA Green Contexts 1 SM*" in the plots – confirm those obtained by using the spatial-regulation based MBMS. This is because the slowdown curves generated using the two regulation approaches are almost coincidental for low SM configurations. Moreover, this regulation approach enables us to unveil the full potential of spatial regulation: reaching the configuration where a single SM produces memory interference, *spmd_traffic_generator_conf_sm* further reduces the

memory interference on *cpu_synth*. This configuration reaches a minimum slowdown of 1.15× on the *NVIDIA AGX Orin* and 1.2× on the *NVIDIA AGX Thor*, demonstrating that *spatial*-driven regulation would be even more effective if GPU partitioning were feasible at the individual SM level.

Evaluation Results, *stride*=512B

Figures 6.11 and 6.12 present the results obtained by configuring the GPU memory traffic generators with $stride = 512B$ on the *NVIDIA AGX Orin* and the *NVIDIA AGX Thor*, respectively. This configuration generates heavily strided memory access pattern from the GPU, requiring each thread to issue a different memory request. This scenario generates low bandwidth consumption for the GPU, as per-request overhead limits the number of transactions at main memory level. However, the main memory is saturated with memory requests. Therefore, this pattern can generate potentially much higher interference than the *coalesced* one.

Figures 6.11 and 6.12 show that the maximum bandwidth reached by the benchmarks in the strided configuration is much lower than in the *coalesced* configuration. Specifically, on *NVIDIA AGX Orin*, the highest bandwidth configuration of *spmd_traffic_generator* is *trf=read* (see Figure 6.11a), which reaches the memory bandwidth of 120GB/s. In the same way, on *NVIDIA AGX Thor*, the highest bandwidth configuration is the *trf=read* one (see Figure 6.12a), reaching a memory bandwidth of 65GB/s.

The slowdown generated by this memory access pattern heavily depends on the platform: on the *NVIDIA AGX Orin* platform the slowdown due to *trf=read* (see Figure 6.11a) is nullified ($< 1.05\times$), meaning that the CPU traffic is not interfered by strided GPU read requests. This occurs for both *spmd_traffic_generator* and *spmd_traffic_generator_conf_sm*, as the two GPU memory traffic generators behave similarly. At the same time, *trf=write* traffic generates a 1.45× slowdown (see Figure 6.11b), which is comparable to the one obtained in the coalesced configuration. By regulating the GPU using the *spatial*-domain MBMS, the slowdown is reduced to an 1.2×. At the same time, by further reducing the number of SMs used

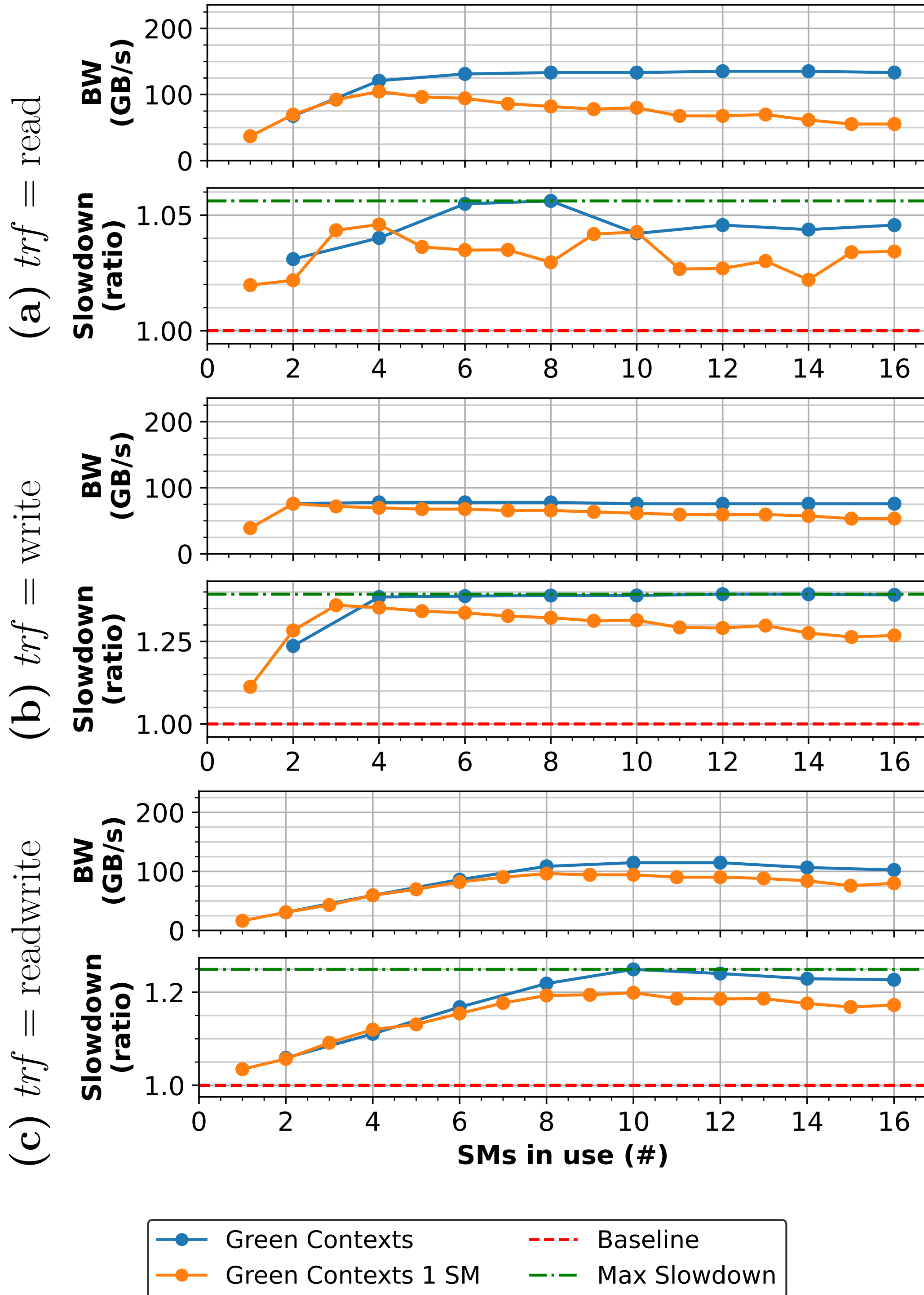


Figure 6.11: Synthetic benchmark results on *NVIDIA AGX Orin*, *strided*

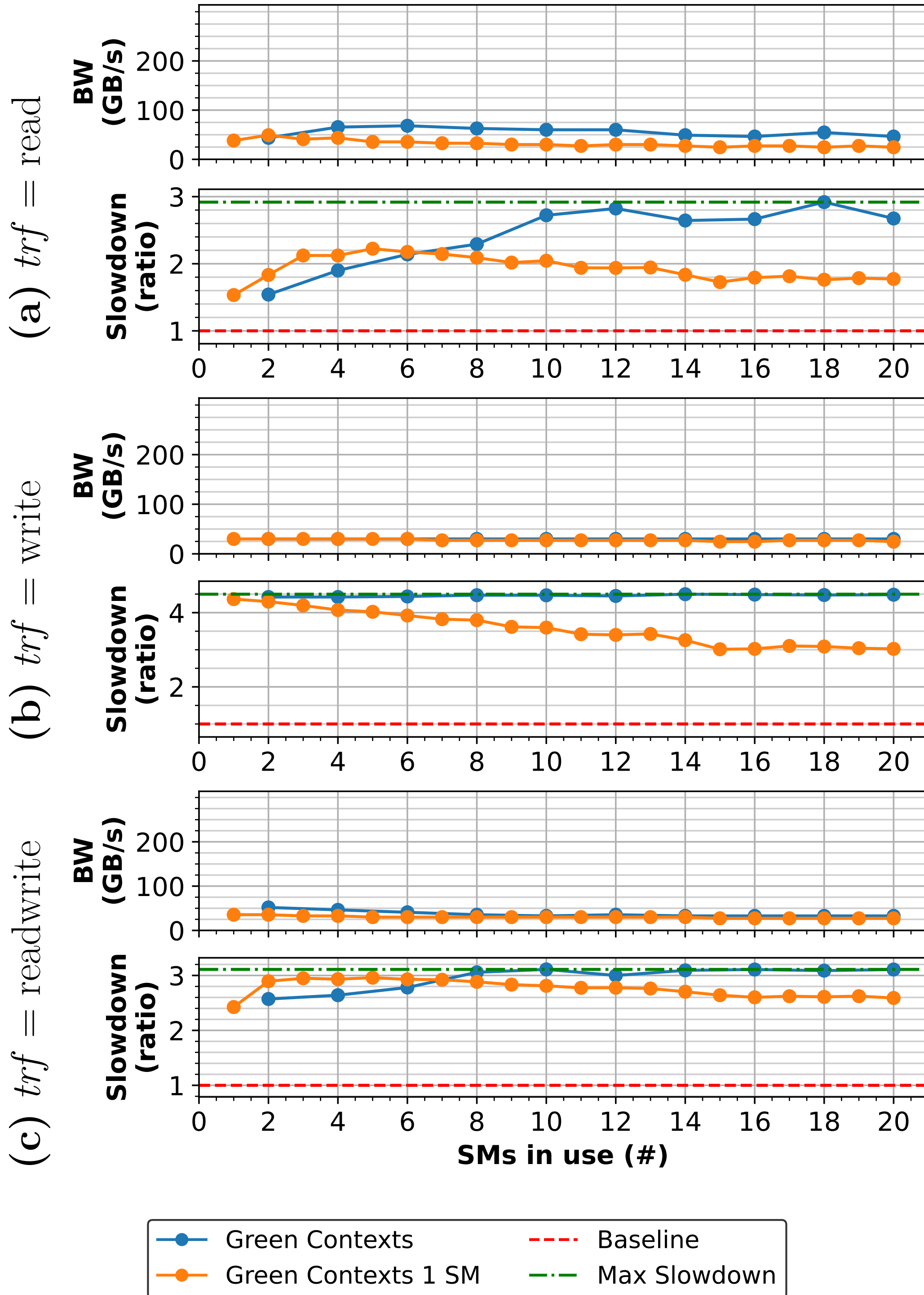


Figure 6.12: Synthetic benchmark results on the *NVIDIA AGX Thor*, *strided*

by *spmd_traffic_generator_conf_sm* to a single SM, the slowdown reaches 1.1×, demonstrating that *spatial-based*-based regulation is also effective for strided memory access patterns.

Unlike the *NVIDIA AGX Orin*, on the *NVIDIA AGX Thor* the slowdown caused by the strided memory access pattern is magnified. In fact, the slowdown caused by *trf=read* strided pattern rises to 3× (see Figure 6.12a), while the slowdown caused by *trf=write* reaches 5× (see Figure 6.12b). By regulating the GPU using the *spatial*-domain based MBMS, the slowdown caused by *trf=read* strided access pattern is reduced to 1.4×, while the one caused by *trf=write* pattern demonstrates to be insensitive to *regulation.* This happens because even a single SM, when performing *trf=write* strided memory accesses, saturates the main memory with memory requests, causing the CPU tasks to suffer additional latency. This phenomenon also occurs when the the GPU kernels perform *trf=readwrite* memory accesses (see Figure 6.12c). In this case, the slowdown caused on the *cpu_synth* benchmark reaches 3.1×. By regulating the *spmd_traffic_generator* benchmark with the MBMS, this slowdown is mitigated down to 2.5×, which indicates limited control.

Ultimately, this result unveils a pathological characteristic of the *NVIDIA AGX Thor* platform: even a single SM executing in the GPU can cause severe memory interference and, in turn, disproportional slowdown to tasks executing on the CPU. Our current work is focusing on better studying this phenomenon.

6.5 Real-world benchmarks evaluation

In this section, we evaluate the *spatial*-based MBMS using the PolyBench-ACC [57] benchmark suite. In the following, we discuss the evaluation setup and the results.

6.5.1 Evaluation Setup

In this evaluation, we use the *cpu_synth* benchmark as *main* task and the PolyBench-ACC benchmarks as *interfering* tasks (one at a time). We compile

Algorithm 11: GPU-to-CPU Memory Interference Measurement in PolyBench-ACC

```
1  total_ops ← 0;
2  Function cpu_synth():
3  |  while polybench_running() do
4  |  |  mem_ops(chunk_size);
5  |  |  total_ops += chunk_size;
6  |  end
7  Function polybench_benchmark():
8  |  start_cpu_synth();
9  |  ...;
10 |  start ← time();
11 |  total_ops ← 0;
12 |  cudaLaunchKernel⟨⟨⟨...⟩⟩⟩(kernel);
13 |  stop ← time();
14 |  store_result(kernel, stop − start, total_ops);
15 |  ...;
```

the PolyBench-ACC benchmarks to make use of the GPU to optimize data-parallel portions of the benchmarks. This way, each PolyBench benchmark can launch multiple kernels (one at a time) on the GPU, generating GPU-to-CPU memory interference.

To evaluate the memory interference generated by individual GPU kernels from the PolyBench-ACC suite, we require precise synchronization between *cpu_synth* execution and the GPU kernels. Algorithm 11 illustrates our synchronization approach. Specifically, we modify the PolyBench-ACC benchmarks to launch *cpu_synth* as a POSIX thread at startup (see Line 8). This thread continuously executes chunks of memory operations (see Line 4) while tracking their count (see Line 5), running until the PolyBench-ACC benchmark completes.

The operation counter is shared between *cpu_synth* and the PolyBench-ACC benchmarks via shared memory (see Line 1). To obtain per-GPU kernel measurements, each PolyBench-ACC benchmark performs the following sequence: starts a timer (see Line 10), resets the shared counter (see Line

11), synchronously executes the GPU kernel (see Line 12), then records both the counter value and kernel execution time (see Line 14). Dividing the kernel execution time by the number of memory operations performed by *cpu_synth* during that interval yields the memory access latency experienced by *cpu_synth* when co-scheduled with the GPU kernel. Note that, differently from the original implementation of *cpu_synth*, this value corresponds to the latency per single memory access. This modification is necessary because the number of memory accesses performed by *cpu_synth* depends on the duration of the GPU kernel, and it is not fixed as in the original implementation. Comparing the *memory access latency* of *cpu_synth* when interfered and in isolation, we retrieve the *memory access latency* slowdown. This slowdown is computed equally – and can be compared – to the slowdown discussed in Section 6.4.4. Thus, it can be used to perform a comparison of the slowdown caused by synthetic GPU memory traffic generators and the PolyBench-ACC GPU kernels.

In this evaluation, we configure the PolyBench-ACC benchmarks to use the *EXTRALARGE_DATASET* (i.e., the largest dataset they can use), to maximize GPU-induced memory interference on *cpu_synth*.

6.5.2 Evaluation Results

In this section, we discuss the evaluation results obtained by co-scheduling the *cpu_synth* benchmark and the PolyBench-ACC benchmarks (one at a time). In this setup, the *spatial*-based MBMS regulates GPU memory interference induced on *cpu_synth*. Specifically, we co-execute *cpu_synth* and the PolyBench-ACC benchmarks, allocating portions of the GPU SMs to PolyBench-ACC GPU kernels. For each combination of PolyBench-ACC benchmark and GPU SMs allocation, we repeat the experiment 10 times, for improved statistical significance.

Figures 6.13, 6.14, 6.15, and 6.16 present the results of this evaluation. Due to the high number of GPU kernels implemented by the PolyBench-ACC benchmark suite, the results are shown in the form of a heatmap, with the GPU kernels reported on the rows, and the GPU SMs partitioning con-

figuration on the columns (in descending order). For each PolyBench-ACC GPU kernel, we report the average duration in isolation, the standard deviation in the form of a percentage, and the number of executions throughout the evaluation. Note that the same GPU kernel can be executed multiple times by the executing Polybench-ACC benchmark. Therefore, the number of execution can be higher than 10.

Figures 6.13 and 6.15 present the slowdown induced by PolyBench-ACC GPU kernels on the *cpu_synth* benchmark. This slowdown represents the *memory access slowdown* described in Section 6.5.1. Figures 6.14 and 6.16 present the slowdown of GPU kernels, which is computed as the time execution increase when the GPU kernel is executed on the partitioned GPU.

Also in this evaluation, the results depend on the hardware platforms. In the following we describe the results we obtain on both our evaluation platforms: the *NVIDIA AGX Orin* and the *NVIDIA AGX Thor*.

NVIDIA AGX Orin

Figures 6.13 and 6.14 present the results obtained on the *NVIDIA AGX Orin* platform. In particular, the heatmap in Figure 6.13 shows that numerous GPU kernels lead to a slowdown of the CPU task up to $\approx 1.4\times$ (highlighted in red in the heatmap), a slowdown value consistent with that caused by synthetic GPU memory traffic generators. This means that real-world GPU kernels from the PolyBench-ACC benchmark suite can generate synthetic-generator-level memory interference. At the same time, as the GPU is *regulated* (i.e., the number of SMs in use is reduced), the GPU-to-CPU memory interference decreases and so does the CPU slowdown values. In the most restrictive *regulation* configuration (i.e., the rightmost one, 2 SMs), the slowdown suffered by *cpu_synth* is reduced below $1.1\times$ for all GPU kernels (the maximum value is highlighted in blue in the heatmap). This ultimately proves the effectiveness of the *space*-based MBMS.

Figure 6.14 presents the slowdown of the GPU kernels as a function of the GPU *regulation* configuration. In this figure, the GPU slowdown values reported in the leftmost column represent the GPU kernel slowdowns caused

CPU Slowdown Ratio

GPU Kernel (duration (ms), stddev (%), repetitions (#)) \ SMs in use (#)	16	14	12	10	8	6	4	2
convolution2D_kernel (18.413ms, 2.1%, 10)	1.32	1.29	1.27	1.24	1.21	1.18	1.13	1.07
mm2_kernel1 (1837.82ms, 0.5%, 10)	1.02	1.01	1.01	1.01	1.01	1.01	1.01	1.0
mm2_kernel2 (1813.868ms, 0.5%, 10)	1.02	1.02	1.02	1.01	1.01	1.01	1.01	1.0
convolution3D_kernel (0.037ms, 70.5%, 5100)	1.17	1.16	1.16	1.15	1.14	1.13	1.11	1.08
mm3_kernel1 (224.038ms, 0.3%, 10)	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.0
mm3_kernel2 (223.335ms, 0.2%, 10)	1.01	1.01	1.01	1.01	1.01	1.01	1.0	1.0
mm3_kernel3 (223.626ms, 0.3%, 10)	1.02	1.01	1.01	1.01	1.01	1.01	1.0	1.0
adi_kernel4 (0.015ms, 5.8%, 40950)	1.03	1.02	1.03	1.02	1.02	1.02	1.02	1.02
adi_kernel6 (0.015ms, 18.8%, 40940)	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02
adi_kernel5 (0.047ms, 36.0%, 10)	1.06	1.07	1.06	1.05	1.07	1.05	1.05	1.05
adi_kernel3 (7.55ms, 8.1%, 10)	1.13	1.09	1.09	1.07	1.06	1.04	1.03	1.02
adi_kernel1 (21.682ms, 10.7%, 10)	1.15	1.1	1.09	1.09	1.1	1.05	1.05	1.03
adi_kernel2 (0.068ms, 42.1%, 10)	1.05	1.09	1.06	1.06	1.08	1.01	1.06	1.04
atax_kernel1 (116.389ms, 0.2%, 10)	1.03	1.03	1.03	1.02	1.02	1.01	1.01	1.01
atax_kernel2 (115.354ms, 0.3%, 10)	1.02	1.02	1.02	1.02	1.01	1.01	1.01	1.0
bicg_kernel1 (19.659ms, 0.3%, 10)	1.14	1.14	1.14	1.09	1.09	1.1	1.07	1.04
bicg_kernel2 (1357.332ms, 0.3%, 10)	1.01	1.01	1.0	1.0	1.0	1.01	1.01	1.01
std_kernel (5.457ms, 0.6%, 10)	1.1	1.11	1.1	1.11	1.1	1.11	1.07	1.06
reduce_kernel (3.841ms, 0.9%, 10)	1.38	1.34	1.3	1.26	1.22	1.18	1.14	1.07
corr_kernel (347684.048ms, 2.3%, 10)	1.02	1.02	1.02	1.01	1.02	1.01	1.02	1.01
mean_kernel (8.002ms, 0.2%, 10)	1.1	1.1	1.1	1.1	1.1	1.1	1.07	1.07
mean_kernel (7.984ms, 0.2%, 10)	1.1	1.09	1.1	1.1	1.1	1.1	1.07	1.07
reduce_kernel (0.989ms, 3.1%, 10)	1.36	1.34	1.3	1.28	1.24	1.21	1.16	1.1
covar_kernel (347908.018ms, 1.9%, 10)	1.02	1.02	1.02	1.01	1.02	1.01	1.02	1.01
doitgen_kernel1 (1.756ms, 2.4%, 5120)	1.01	1.01	1.01	1.01	1.01	1.01	1.0	1.0
doitgen_kernel2 (0.019ms, 16.2%, 5120)	1.06	1.07	1.06	1.07	1.07	1.07	1.07	1.05
fdtd_step1_kernel (5.023ms, 3.3%, 5000)	1.35	1.33	1.3	1.26	1.22	1.18	1.13	1.08
fdtd_step2_kernel (5.025ms, 2.6%, 5000)	1.35	1.33	1.3	1.26	1.22	1.18	1.13	1.08
fdtd_step3_kernel (6.28ms, 2.2%, 5000)	1.36	1.34	1.32	1.27	1.24	1.19	1.14	1.08
gemm_kernel (227.575ms, 0.2%, 10)	1.02	1.02	1.01	1.01	1.01	1.01	1.01	1.01
gemver_kernel1 (16.619ms, 1.7%, 10)	1.36	1.33	1.3	1.26	1.23	1.19	1.15	1.08
gemver_kernel2 (19.172ms, 0.3%, 10)	1.15	1.14	1.14	1.08	1.09	1.08	1.05	1.02
gemver_kernel3 (1358.482ms, 0.2%, 10)	1.01	1.0	1.0	1.0	1.0	1.0	1.01	1.01
gesummv_kernel (8541.307ms, 0.1%, 10)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
gramschmidt_kernel1 (2.368ms, 4.8%, 81920)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
gramschmidt_kernel2 (0.161ms, 1.3%, 81920)	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.01
gramschmidt_kernel3 (15.724ms, 6.0%, 81920)	1.06	1.06	1.06	1.06	1.06	1.06	1.05	1.03
runJacobiCUDA_kernel1 (0.015ms, 39.9%, 100000)	1.01	1.02	1.01	1.02	1.01	1.02	1.02	1.01
runJacobiCUDA_kernel2 (0.015ms, 3.9%, 100000)	1.01	1.02	1.01	1.02	1.01	1.02	1.02	1.01
runJacobiCUDA_kernel1 (1.044ms, 15.2%, 200)	1.33	1.31	1.28	1.25	1.21	1.17	1.13	1.07
runJacobiCUDA_kernel2 (0.943ms, 1.7%, 200)	1.38	1.35	1.32	1.28	1.23	1.18	1.13	1.08
mvt_kernel1 (116.429ms, 0.3%, 10)	1.03	1.03	1.02	1.02	1.02	1.01	1.01	1.01
mvt_kernel2 (115.239ms, 0.3%, 10)	1.02	1.02	1.02	1.01	1.01	1.01	1.01	1.0
syr2k_kernel (7235.116ms, 0.0%, 10)	1.02	1.01	1.01	1.01	1.01	1.01	1.01	1.0
syrk_kernel (3723.373ms, 0.4%, 10)	1.01	1.01	1.01	1.01	1.01	1.0	1.0	1.0

Figure 6.13: GPU slowdown results obtained using PolyBench-ACC benchmarks on the *NVIDIA AGX Orin* platform.

GPU Slowdown Ratio

GPU Kernel (duration (ms), stddev (%), repetitions (#))	16	14	12	10	8	6	4	2
convolution2D_kernel (18.413ms, 2.1%, 10)	1.0	1.1	1.2	1.4	1.6	2.0	2.8	5.3
mm2_kernel1 (1837.82ms, 0.5%, 10)	1.0	1.1	1.3	1.6	2.0	2.6	4.0	7.9
mm2_kernel2 (1813.868ms, 0.5%, 10)	1.0	1.1	1.3	1.6	2.0	2.6	4.0	7.9
convolution3D_kernel (0.037ms, 70.5%, 5100)	1.4	1.4	1.4	1.5	1.7	1.9	2.5	3.8
mm3_kernel1 (224.038ms, 0.3%, 10)	1.0	1.1	1.3	1.6	2.0	2.7	4.0	7.9
mm3_kernel2 (223.335ms, 0.2%, 10)	1.0	1.1	1.3	1.6	2.0	2.7	4.0	8.0
mm3_kernel3 (223.626ms, 0.3%, 10)	1.0	1.1	1.3	1.6	2.0	2.7	4.0	7.9
adi_kernel4 (0.015ms, 5.8%, 40950)	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
adi_kernel6 (0.015ms, 18.8%, 40940)	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
adi_kernel5 (0.047ms, 36.0%, 10)	1.6	2.1	1.6	2.0	1.8	1.9	1.7	2.1
adi_kernel3 (7.55ms, 8.1%, 10)	0.9	1.5	1.6	2.0	2.6	3.2	3.3	5.6
adi_kernel1 (21.682ms, 10.7%, 10)	1.0	1.1	1.3	1.5	1.9	3.0	3.2	5.5
adi_kernel2 (0.068ms, 42.1%, 10)	1.4	1.7	1.4	1.8	1.5	1.8	1.5	2.1
atax_kernel1 (116.389ms, 0.2%, 10)	1.0	1.2	1.3	1.6	2.0	2.7	4.0	7.9
atax_kernel2 (115.354ms, 0.3%, 10)	1.0	1.2	1.3	1.5	1.8	2.5	3.7	7.2
bicg_kernel1 (19.659ms, 0.3%, 10)	1.0	1.0	1.0	2.0	2.0	2.0	2.9	5.9
bicg_kernel2 (1357.332ms, 0.3%, 10)	1.0	1.0	1.3	1.7	1.7	1.0	0.6	0.1
std_kernel (5.457ms, 0.6%, 10)	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
reduce_kernel (3.841ms, 0.9%, 10)	1.0	1.1	1.3	1.5	1.8	2.3	3.3	6.4
corr_kernel (347684.048ms, 2.3%, 10)	1.0	1.0	1.1	1.1	1.2	1.4	0.7	1.3
mean_kernel (8.002ms, 0.2%, 10)	1.0	1.0	1.0	1.0	1.0	1.0	1.7	2.4
mean_kernel (7.984ms, 0.2%, 10)	1.0	1.0	1.0	1.0	1.0	1.0	1.7	2.4
reduce_kernel (0.989ms, 3.1%, 10)	1.0	1.1	1.2	1.4	1.6	2.1	3.0	5.6
covar_kernel (347908.018ms, 1.9%, 10)	1.0	1.0	1.1	1.1	1.2	1.4	0.7	1.3
doitgen_kernel1 (1.756ms, 2.4%, 5120)	1.0	1.0	1.1	1.1	1.5	1.7	2.6	4.9
doitgen_kernel2 (0.019ms, 16.2%, 5120)	1.6	1.7	1.9	2.0	2.0	2.1	2.4	3.5
fdtd_step1_kernel (5.023ms, 3.3%, 5000)	1.0	1.1	1.1	1.3	1.5	1.8	2.6	4.8
fdtd_step2_kernel (5.025ms, 2.6%, 5000)	1.0	1.1	1.1	1.3	1.5	1.8	2.6	4.8
fdtd_step3_kernel (6.28ms, 2.2%, 5000)	1.0	1.0	1.1	1.2	1.3	1.6	2.3	4.1
gemm_kernel (227.575ms, 0.2%, 10)	1.0	1.1	1.3	1.6	2.0	2.7	4.0	7.9
gemver_kernel1 (16.619ms, 1.7%, 10)	1.0	1.1	1.2	1.4	1.6	2.1	2.9	5.6
gemver_kernel2 (19.172ms, 0.3%, 10)	1.0	1.0	1.0	2.0	2.0	2.0	3.0	6.0
gemver_kernel3 (1358.482ms, 0.2%, 10)	1.0	1.0	1.4	2.1	1.9	1.1	0.6	0.1
gesummv_kernel (8541.307ms, 0.1%, 10)	1.0	1.2	1.4	1.6	1.5	1.5	1.6	1.4
gramschmidt_kernel1 (2.368ms, 4.8%, 81920)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
gramschmidt_kernel2 (0.161ms, 1.3%, 81920)	1.1	1.2	1.1	1.2	1.1	1.4	1.5	1.9
gramschmidt_kernel3 (15.724ms, 6.0%, 81920)	1.0	1.0	1.0	1.0	1.0	1.0	1.3	1.9
runJacobiCUDA_kernel1 (0.015ms, 39.9%, 100000)	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
runJacobiCUDA_kernel2 (0.015ms, 3.9%, 100000)	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
runJacobiCUDA_kernel1 (1.044ms, 15.2%, 200)	1.0	1.1	1.2	1.4	1.6	2.1	2.9	5.4
runJacobiCUDA_kernel2 (0.943ms, 1.7%, 200)	1.0	1.1	1.2	1.4	1.6	2.1	3.0	5.7
mvt_kernel1 (116.429ms, 0.3%, 10)	1.0	1.2	1.3	1.6	2.0	2.7	4.0	7.9
mvt_kernel2 (115.239ms, 0.3%, 10)	1.0	1.2	1.3	1.5	1.8	2.5	3.7	7.2
syr2k_kernel (7235.116ms, 0.0%, 10)	1.0	1.2	1.3	1.6	2.0	2.7	4.1	8.2
syrk_kernel (3723.373ms, 0.4%, 10)	1.0	1.1	1.3	1.6	2.0	2.7	4.0	8.1

SMs in use (#)

Figure 6.14: GPU slowdown results obtained using PolyBench-ACC benchmarks on the *NVIDIA AGX Orin* platform.

by the MBMS when regulation is not active. Therefore, this represents the overhead introduced by the MBMS on GPU kernels. The highest value in this column reaches $\approx 2\times$ (highlighted in red). However, this slowdown value corresponds to very short GPU kernels. Specifically, the GPU kernels that suffer $\approx 2\times$ slowdown run for $15us$. This means that the *space*-based MBMS introduces $\approx 7.5us$ overhead per GPU kernel launch.

As the *regulation* becomes more strict, the GPU kernel execution times change in different ways. As expected, for some kernels, the execution times increases. In those cases, the GPU kernels slowdown reaches $\approx 8\times$, which is consistent with the GPU SMs reduction ratio (16 to 2 SMs). In a very specific case (*gramschmidt_kernel1*), the execution time is unchanged throughout the range of GPU regulation. This can happen if the GPU kernel is undersized for the GPU, and all the GPU threads can be executed on a single SM. In other cases, the GPU kernel speeds up when *regulated* (*bicg_kernel2*, *gemver_kernel3*). This phenomenon is unexpected and can be caused by some caching effect. In fact, by forcing all GPU threads to execute on two SMs, all threads access the same L1 Data cache on the GPU. This can cause the reported speedup. However, at the time of writing, CUDA does not support GPU kernel profiling when using CUDA Green Contexts on the *NVIDIA AGX Orin*. For this reason, it is currently impossible to analytically study the cause of this speedup. A systematic analysis of this phenomenon is left for future work.

NVIDIA AGX Thor

Figures 6.15 and 6.16 report the CPU and GPU slowdown results obtained on the *NVIDIA AGX Thor* platform. In particular, Figure 6.15 shows that, on this platform, the GPU kernels from the PolyBench-ACC suite can cause up to $\approx 2.2\times$ slowdown of the *cpu_synth* task (highlighted in red). These slowdown results are consistent to the ones discussed for the synthetic benchmark evaluation. As *regulation* becomes more restrictive, the slowdown values decrease. In the most *regulated* configuration (2 SMs), the highest reported slowdown is $\approx 1.5\times$ (highlighted in blue), also consistent with the previous

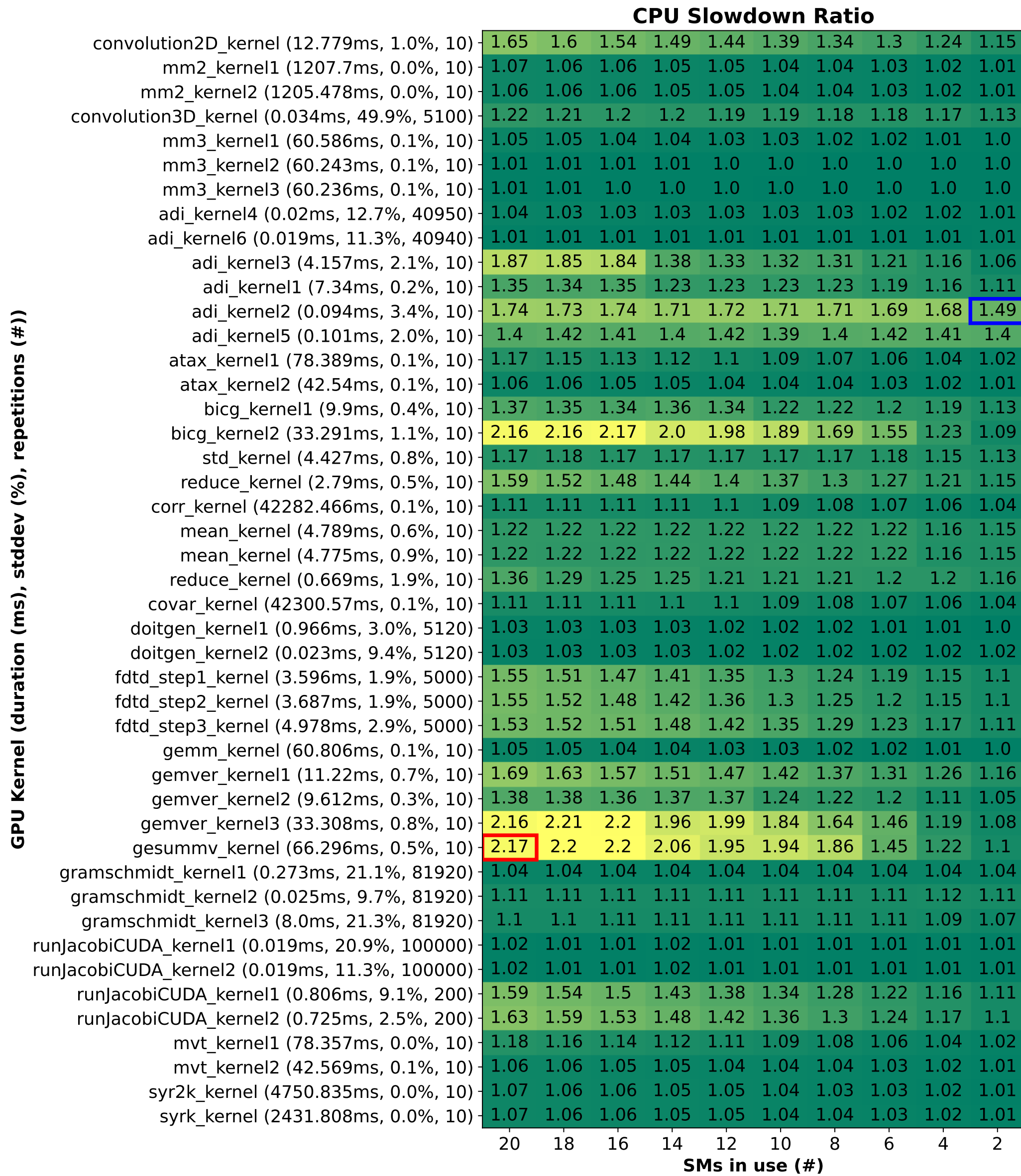


Figure 6.15: CPU slowdown results obtained using PolyBench-ACC benchmarks on the *NVIDIA AGX Thor* platform.

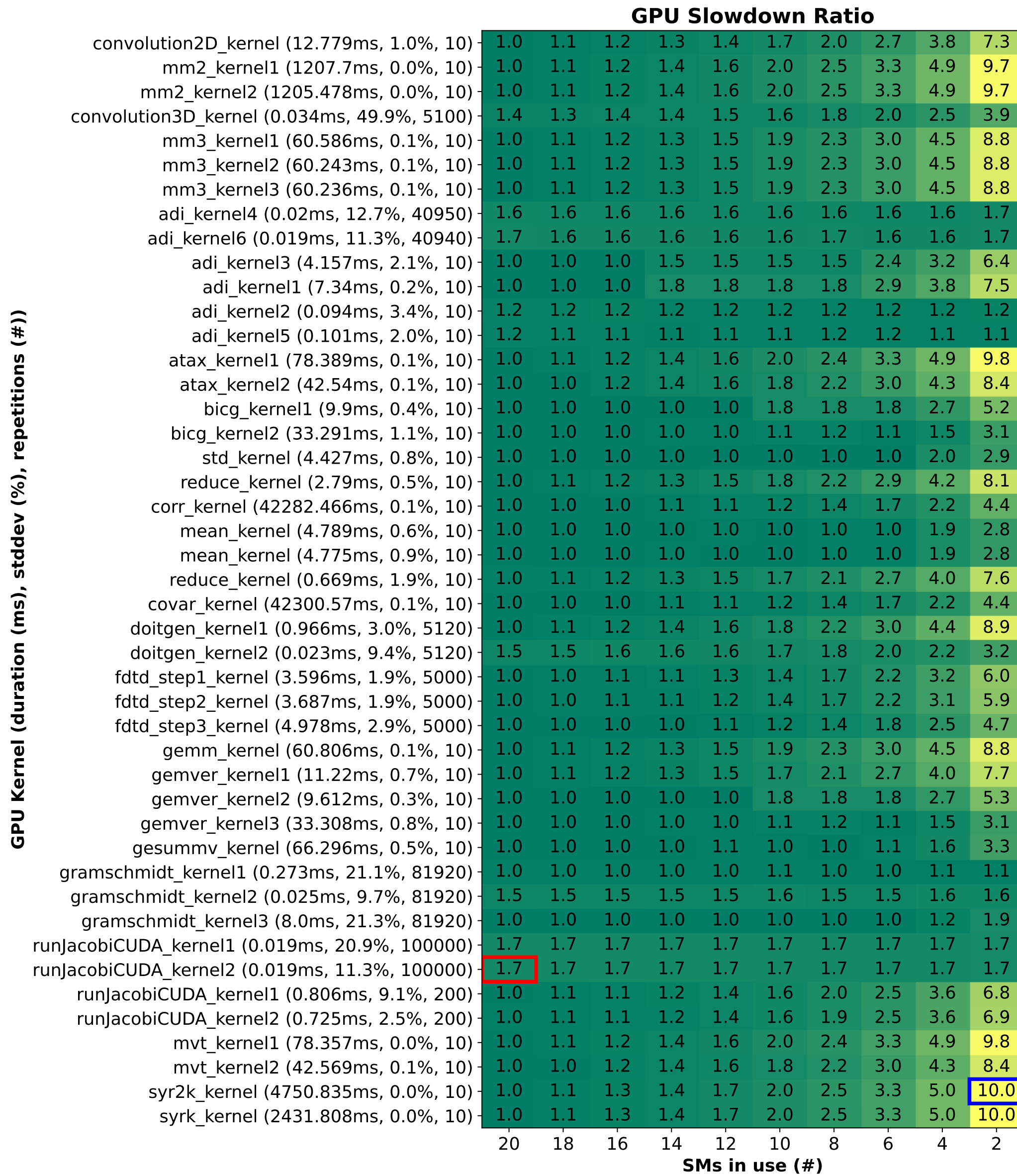


Figure 6.16: GPU slowdown results obtained using PolyBench-ACC benchmarks on the *NVIDIA AGX Thor* platform.

results.

In the very specific case of the *adi_kernel5* kernel, the GPU-to-CPU memory interference is not controlled by the MBMS, as the slowdown of *cpu_synth* is unvaried in all GPU SM partitioning configurations. The *adi_kernel5* GPU kernel performs a write contiguous memory access pattern (consecutive GPU threads work on consecutive memory location). In the synthetic benchmark evaluation, the same access pattern causes up to 2.5× slowdown of *cpu_synth*, and the slowdown is reduced to 1.6× by the MBMS (see Figure 6.10b). The slowdown caused by *adi_kernel5* in the unregulated configuration (i.e., 20SMs, 1.5× slowdown) is lower than the slowdown generated by the synthetic GPU memory traffic generator in the maximally regulated configuration (i.e., 1.6×). This means that *adi_kernel5* is severely undersized for the GPU, and spatial-regulation has no effect on its performance. For this reason, the slowdown it generates is not controlled by the MBMS. Ultimately, these results confirm the ones obtained in the synthetic benchmark evaluation.

Figure 6.16 presents the GPU slowdown results obtained on the *NVIDIA AGX Thor* platform. The results indicate a slowdown of up to $\approx 1.7\times$ when the regulation is not active (highlighted in red). These slowdown values correspond to the GPU kernels that run for $\approx 19us$ (*runJacobiCUDA_kernel1*, *runJacobiCUDA_kernel2*). Therefore, the overhead introduced by the MBMS corresponds to $\approx 7.5us$ per GPU kernel launch. As GPU regulation becomes more restrictive, the execution time of GPU kernels is affected in different ways. As observed on the *NVIDIA AGX Orin* platform, the execution time of the *gramschmidt_kernel1* kernel is unchanged through all GPU partitioning configurations. In other cases, as expected, the execution time of the GPU kernel increases. On this platform, GPU slowdown values reach $\approx 10\times$ (highlighted in blue), which is proportional to the SM reduction ratio (20 SMs to 2 SMs).

Limitation of the Analysis and Future Work

Figures 6.13, 6.14, 6.15, and 6.16 show elevated standard deviation for certain GPU kernels. These high values correlate with short-duration kernels, whose execution time is comparable to the MBMS overhead. The overhead of the MBMS is primarily caused by the *worker thread*, which requires asynchronous signal operations to launch GPU kernels (see Section 6.2.2). This mechanism introduces inherent result variability. Consequently, increasing the number of repetitions of the PolyBench-ACC benchmarks would not improve stability.

However, the use of a dedicated working thread is necessary to maintain the asynchronous execution of the GPU kernels. In fact, the only way to avoid the use of a separate thread would be by implementing synchronous kernel launches operation, where GPU-accelerated tasks actively wait for the completion of GPU kernels. This would break the execution model proposed by CUDA, where the CPU and the GPU are designed to execute concurrently.

Future work will focus on better characterizing the source of this instability, trying to optimize the use the working thread to launch GPU kernels. In turn, this will lead to more stable results.

6.6 Conclusion

In this chapter, we presented and evaluated a memory bandwidth management scheme based on *spatial*-domain *regulation*. To the best of our knowledge, this represents the first attempt in regulating the memory bandwidth consumed by a GPU acting in the *spatial*-domain. Specifically, the MBMS we proposed limits the number of SMs the GPU can use to execute kernels, in turn limiting the memory bandwidth consumption, and the memory interference induced on the CPU complex.

We evaluated the MBMS using a combination of synthetic and real-world workloads on the two latest embedded platforms provided by NVIDIA: the *NVIDIA AGX Orin* and the *NVIDIA AGX Thor*. The evaluation results show that spatial-regulation can effectively mitigate the memory interference induced by the GPU for most of the GPU memory traffics. Moreover, this

analysis unveils a previously unknown phenomenal architecture: even a single SM on the *NVIDIA AGX Thor* platform can saturate the main memory, in turn causing memory interference and disproportional slowdown to tasks executing on the CPU. This phenomenon is not controlled by our MBMS, and therefore requires the integration of other mitigation techniques.

Our current work is focusing on integrating time-domain and space-domain technique to mitigate GPU-to-CPU memory interference. This integration can help enforce strict timing constraints on GPU task while ensuring better utilization of the whole hardware.

Chapter 7

Conclusions and Future Work

This thesis has addressed the critical problem of memory interference in Commercial Off-The-Shelf (COTS) Heterogeneous Systems-on-Chip (HeSoCs), which represent the industry standard for deploying Cyber-Physical Systems (CPSs). The research combined detailed empirical characterization of memory interference phenomena with the analysis and design of novel mitigation strategies, contributing to the foundation for designing predictable systems through coordinated hardware-software memory management.

7.1 Contribution Summary

The work made three substantive contributions to the field. First, the thesis challenged a widely held assumption in the literature that main memory bandwidth saturation, induced by synthetic read-intensive workloads with 100% cache miss rates, represents the primary source of worst-case slowdowns among memory interference causes. Through systematic evaluation on representative COTS platforms (Xilinx UltraScale and NVIDIA Tegra TX2), the research demonstrated that interference mechanisms operate differently at distinct levels of the memory hierarchy – including main memory bandwidth saturation, cache interference, and micro-architectural phenomena within CPU cores. By jointly analyzing these factors, the work provided an integrated perspective that cannot be obtained when studying each cause

in isolation. The evaluation revealed that platform-specific characteristics fundamentally shape interference patterns, with different types of interference impacting performance differently and highlighting contention points that limit system scalability.

Second, the thesis addressed a previously understudied aspect of Memory Bandwidth Management Schemes (MBMSs) – the reconfiguration component. While prior work extensively investigated bandwidth regulation techniques, the reconfiguration overhead and latency determining the MBMS effectiveness had received limited attention. This work presented the first in-depth comparative analysis of synchronous and asynchronous reconfiguration techniques on HeSoC platforms. The evaluation, conducted on Xilinx Zynq UltraScale+ with both synthetic and real-world benchmarks from the PolyBench suite, demonstrated that asynchronous techniques improve control granularity by up to 19× compared to synchronous approaches, reducing response times from the millisecond to microsecond scale. This finding has significant implications for dynamic mixed-criticality systems where tasks with varying criticality frequently enter and exit the system.

Third, the thesis extended memory interference mitigation to heterogeneous accelerators, specifically addressing the understudied spatial-domain regulation approach for GPUs. The research proposed a novel MBMS that protects CPU execution from GPU-induced memory interference by dynamically limiting the number of Streaming Multiprocessors (SMs) available to GPU kernels based on CPU workload criticality. The approach leverages CUDA Green Contexts, a NVIDIA-native technology, to enable portable, transparent, and runtime-adaptive control of GPU resource consumption. Extensive evaluation on the latest NVIDIA HeSoCs (AGX Orin and AGX Thor) demonstrated that spatial-domain SM regulation effectively reduces GPU-induced slowdowns across diverse workload configurations. Critically, the evaluation also uncovered a pathological case on the AGX Thor platform: a single SM executing high-stride write operations can saturate the memory controller, inducing up to 5× slowdown on CPU cores. This finding highlights the platform-specific nature of memory interference and the need for adaptive mitigation strategies.

7.2 Limitations and Future Work

Three key limitations define the scope of future work. First, Chapter 4 focused exclusively on the host complex in isolation, studying memory interference effects among CPU cores on Xilinx UltraScale and NVIDIA Tegra TX2 platforms. A deeper investigation must extend this analysis to encompass the broader heterogeneous landscape by integrating more modern platforms with their diverse architectural configurations, and critically, by systematically characterizing interference when both the host complex and accelerator complexes operate concurrently. This extension should cover standard accelerators (FPGAs and GPGPUs) as well as emerging accelerator types, including AI accelerators, Data Processing Units (DPUs), and other domain-specific processors. Only through such comprehensive, multi-complex characterization can we fully understand the memory interference patterns that arise in real heterogeneous deployments.

Second, the bandwidth reconfiguration analysis in Chapter 5 concentrated on a simplified mixed-criticality model featuring a single privileged critical task alongside best-effort background tasks. This constrained model, while enabling focused analysis of reconfiguration techniques, does not reflect the complexity of real mixed-criticality systems that frequently support multiple critical tasks with varying requirements and execution patterns. Extending this work to encompass more general models with multiple privileged tasks is essential for practical applicability. Such extensions should address challenges including coordinating bandwidth allocation across multiple critical tasks, handling priority inversions and fairness concerns among critical tasks, and designing reconfiguration algorithms that scale efficiently with increasing numbers of privileged tasks while maintaining tight control granularity.

Third, the spatial-domain GPU bandwidth management approach proposed in Chapter 6 effectively mitigates GPU-induced interference under typical workload conditions through SM partitioning. However, the discovery of the pathological case on the NVIDIA AGX Thor platform – where even a single SM executing high-stride write operations saturates the memory controller – demonstrates that spatial-domain regulation alone cannot achieve

complete isolation. The essential future work for Chapter 6 is to develop hybrid approaches that seamlessly integrate spatial-domain and temporal-domain regulation. A unified framework should dynamically detect pathological memory access patterns at runtime, classify them according to their interference potential, and adaptively switch between or combine spatial partitioning and temporal throttling mechanisms. Only through such adaptive, hybrid regulation can the system maintain effective isolation even under worst-case GPU memory access patterns.

7.3 Practical Implications

The findings of this thesis have practical implications for several communities. For embedded systems designers, this work provides concrete evidence and tools for characterizing memory interference in target platforms, enabling them to make informed decisions about task mapping and resource allocation. The comparison of synchronous vs. asynchronous reconfiguration offers practical guidance for implementing memory bandwidth management in mixed-criticality systems. For the real-time systems community, the thesis contributes to the foundation for predictable execution on heterogeneous platforms – a critical requirement for safety-critical CPSs. The detailed characterization and proposed mitigation strategies advance the state-of-the-art in real-time heterogeneous computing, reducing the conservatism in schedulability analysis. For hardware manufacturers, the identification of platform-specific phenomena (particularly the AGX Thor pathological case) provides valuable feedback for hardware design, and future platforms can incorporate lessons learned regarding memory controller arbitration, interference prevention, and configurability.

7.4 Concluding Remarks

Ultimately, this thesis tackled the multifaceted problem of memory interference in COTS HeSoCs by combining empirical characterization, analysis of

system components, and design of novel mitigation techniques. The work demonstrated that effective interference control requires a holistic, platform-specific, and multi-level approach. Rather than seeking universal solutions, the research emphasized the importance of understanding each platform's unique characteristics and adapting mitigation strategies accordingly. The three main contributions—detailed multi-level interference characterization, analysis of reconfiguration techniques, and spatial-domain GPU bandwidth management—collectively establish a foundation for designing predictable, mixed-criticality systems on heterogeneous platforms. As Cyber-Physical Systems become increasingly prevalent and heterogeneous computing continues to evolve, the importance of predictable memory subsystem behavior will only grow. This thesis represents a step toward that goal, but substantial work remains. Through the continued application of the methodologies and insights presented here, and through pursuit of the research directions outlined, the community can move toward the ultimate objective: reliable, predictable, high-performance heterogeneous computing for safety-critical applications.

Bibliography

[1] José María Aceituno, Ana Guasque, Patricia Balbastre, José Simó, and Alfons Crespo. Hardware resources contention-aware scheduling of hard real-time multiprocessor systems. *Journal of Systems Architecture*, 118:102223, 2021.

[2] Homa Aghilinasab, Waqar Ali, Heechul Yun, and Rodolfo Pellizzoni. Dynamic memory bandwidth allocation for real-time gpu-based soc platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3348–3360, 2020.

[3] Ankit Agrawal, Gerhard Fohler, Johannes Freitag, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:22, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[4] Ankit Agrawal, Renato Mancuso, Rodolfo Pellizzoni, and Gerhard Fohler. Analysis of dynamic memory bandwidth regulation in multi-core real-time systems. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 230–241, 2018.

[5] Alif Ahmed and Kevin Skadron. Hopscotch: a micro-benchmark suite for memory performance evaluation. In *Proceedings of the International*

Symposium on Memory Systems, MEMSYS '19, page 167–172, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: A predictable sdram memory controller. In *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256, 2007.

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

[8] Waqar Ali and Heechul Yun. Protecting Real-Time GPU Kernels on Integrated CPU-GPU SoC Platforms (Artifact). *Dagstuhl Artifacts Series*, 4(2):3:1–3:2, 2018.

[9] Waqar Ali and Heechul Yun. Rt-gang: Real-time gang scheduling framework for safety-critical systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 143–155, 2019.

[10] Amba 5 homoepage, 2025. Accessed: 2025-12-15.

[11] AMD. *Zynq UltraScale+ Device Technical Reference Manual*, 2015.

[12] Björn Andersson, Arvind Easwaran, and Jinkyu Lee. Finding an upper bound on the increase in execution time due to contention on the memory bus in cots-based multicore systems. *SIGBED Rev.*, 7(1), January 2010.

[13] BJörn Andersson, Hyoseung Kim, Dionisio De Niz, Mark Klein, Ragunathan (Raj) Rajkumar, and John Lehoczky. Schedulability analysis of tasks with corunner-dependent execution times. *ACM Trans. Embed. Comput. Syst.*, 17(3), May 2018.

[14] Matteo Andreozzi, Antonio Frangioni, Laura Galli, Giovanni Stea, and Raffaele Zippo. A milp approach to dram access worst-case analysis. *Computers & Operations Research*, 143:105774, 2022.

[15] ARM. Arm corelink qos-400 network interconnect advanced quality of service supplement to arm corelink nic-400 network interconnect technical reference manual. https://developer.arm.com/documentation/dsu0026/latest/. Accessed: 2025-12-09.

[16] ARM. Arm shared cache partitioning using mpam. https://developer.arm.com/documentation/107804/0002/L3-cache/L3-cache-partitioning. Accessed: 2025-12-07.

[17] ARM. Arm a53 documentation, 2025. Accessed: 2024-06-21.

[18] ARM. *Arm Cortex-A53 MPCore Processor Technical Reference Manual r0p4*, 2025.

[19] Joshua Bakita and James H. Anderson. Hardware Compute Partitioning on NVIDIA GPUs for Composable Systems. In Renato Mancuso, editor, *37th Euromicro Conference on Real-Time Systems (ECRTS 2025)*, volume 335 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:25, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[20] Ayoosh Bansal, Rohan Tabish, Giovani Gracioli, Renato Mancuso, Rodolfo Pellizzoni, and Marco Caccamo. Evaluating memory subsystem of configurable heterogeneous mpsoc. *Proceedings of the Operating Systems Platforms for Embedded Real-Time applications*, 2018.

[21] Javier Barrera, Leonidas Kosmidis, Hamid Tabani, Enrico Mezzetti, Jaume Abella, Mikel Fernandez, Guillem Bernat, and Francisco J. Cazorla. On the reliability of hardware event monitors in mpsocs for critical domains. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 580–589, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Michael Bechtel and Heechul 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)*, pages 357–367, 2019.

[23] Michael Bechtel and Heechul Yun. Denial-of-service attacks on shared resources in intel's integrated cpu-gpu platforms. In *2022 IEEE 25th International Symposium On Real-Time Distributed Computing (ISORC)*, pages 1–9, 2022.

[24] Michael Bechtel and Heechul Yun. Analysis and mitigation of shared resource contention on heterogeneous multicore: An industrial case study. *IEEE Transactions on Computers*, 73(7):1753–1766, 2024.

[25] Gianluca Bellocchi, Alessandro Capotondi, Francesco Conti, and Andrea Marongiu. A risc-v-based fpga overlay to simplify embedded accelerator deployment. In *2021 24th Euromicro Conference on Digital System Design (DSD)*, pages 9–17, 2021.

[26] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, 2002.

[27] Alessandro Biondi, Daniel Casini, Giorgiomaria Cicero, Niccolò Borgioli, Giorgio Buttazzo, Gaetano Patti, Luca Leonardi, Lucia Lo Bello, Marco Solieri, Paolo Burgio, Ignacio Sanudo Olmedo, Angelo Ruocco, Luca Palazzi, Marko Bertogna, Alessandro Cilardo, Nicola Mazzocca, and Antonino Mazzeo. Sphere: A multi-soc architecture for next-generation cyber-physical systems based on heterogeneous platforms. *IEEE Access*, 9:75446–75459, 2021.

[28] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4), December 2010.

[29] Jalil Boudjadar, Jin Hyun Kim, and Simin Nadjm-Tehrani. Performance-aware scheduling of multicore time-critical systems. In

ACM/IEEE International Conference on Formal Methods and Models for System Design, page 105 – 114, 2016.

[30] G. Brilli, G. Valente, A. Capotondi, P. Burgio, T. Di Masciov, P. Valente, and A. Marongiu. Fine-grained qos control via tightly-coupled bandwidth monitoring and regulation for fpga-based heterogeneous socs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.

[31] Gianluca Brilli and Paolo Burgio. Interference analysis of shared last-level cache on embedded gp-gpus with multiple cuda streams. *arXiv preprint arXiv:2310.04848*, 2023.

[32] Gianluca Brilli, Alessandro Capotondi, Paolo Burgio, and Andrea Marongiu. Understanding and mitigating memory interference in fpga-based hesocs. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1335–1340, 2022.

[33] Gianluca Brilli, Roberto Cavicchioli, Marco Solieri, Paolo Valente, and Andrea Marongiu. Evaluating controlled memory request injection for efficient bandwidth utilization and predictable execution in heterogeneous socs. *ACM Trans. Embed. Comput. Syst.*, 22(1), December 2022.

[34] Gianluca Brilli, Giacomo Valente, Alessandro Capotondi, Tania Di Mascio, and Andrea Marongiu. Invited Paper: On the Granularity of Bandwidth Regulation in FPGA-Based Heterogeneous Systems on Chip. In Thomas Carle, editor, *22nd International Workshop on Worst-Case Execution Time Analysis (WCET 2024)*, volume 121 of *Open Access Series in Informatics (OASIcs)*, pages 5:1–5:11, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[35] Paolo Burgio, Marko Bertogna, Ignacio Sañudo Olmedo, Paolo Gai, Andrea Marongiu, and Michal Sojka. A software stack for next-generation automotive systems on many-core heterogeneous platforms. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 55–59, 2016.

[36] Nicola Capodieci, Roberto Cavicchioli, Ignacio Sañudo Olmedo, Marco Solieri, and Marko 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)*, pages 1–10, 2020.

[37] Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. Sigamma: server based integrated gpu arbitration mechanism for memory accesses. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, RTNS '17, page 48–57, New York, NY, USA, 2017. Association for Computing Machinery.

[38] Jordi Cardona, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Maximum-contention control unit (mccu): Resource access count and contention time enforcement. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 710–715, 2019.

[39] Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–10, 2017.

[40] Roberto Cavicchioli, Nicola Capodieci, Marco Solieri, Marko Bertogna, Paolo Valente, and Andrea Marongiu. Evaluating controlled memory request injection to counter prem memory underutilization. In *Job Scheduling Strategies for Parallel Processing: 23rd International Workshop, JSSPP 2020, New Orleans, LA, USA, May 22, 2020, Revised Selected Papers*, page 85–105, Berlin, Heidelberg, 2020. Springer-Verlag.

[41] Hoon Sung Chwa, Kang G Shin, Hyeongboo Baek, and Jinkyu Lee. Physical-state-aware dynamic slack management for mixed-criticality systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 129–139. IEEE, 2018.

[42] Neal C. Crago, Mark Stephenson, and Stephen W. Keckler. Exposing memory access patterns to improve instruction and memory efficiency in gpus. *ACM Trans. Archit. Code Optim.*, 15(4), October 2018.

[43] Cuda contexts, 2025. Accessed: 2025-12-15.

[44] Frederica Darema. Spmd computational model. In *Encyclopedia of Parallel Computing*, pages 1933–1943. Springer, 2011.

[45] Dakshina Dasari, Benny Akesson, Vincent Nélis, Muhammad Ali Awan, and Stefan M. Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 39–48, 2013.

[46] Julien Durand, Youcef Bouchebaba, and Luca Santinelli. Statistical analysis for shared resources effects with multi-core real-time systems. In *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 362–371, 2019.

[47] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 104–117, 2018.

[48] Jianbin Fang, Chun Huang, Tao Tang, and Zheng Wang. Parallel programming models for heterogeneous many-cores: a comprehensive survey. *CCF Transactions on High Performance Computing*, 2(4):382–400, 2020.

[49] Giorgio Farina, Gautam Gala, Marcello Cinque, and Gerhard Fohler. Assessing intel's memory bandwidth allocation for resource limitation in real-time systems. In *2022 IEEE 25th International Symposium On Real-Time Distributed Computing (ISORC)*, pages 1–8, 2022.

[50] Farzad Farshchi, Qijing Huang, and Heechul Yun. Bru: Bandwidth regulation unit for real-time multicore processors. In *2020 IEEE Real-*

Time and Embedded Technology and Applications Symposium (RTAS), pages 364–375, 2020.

[51] Jonas Flodin, Kai Lampka, and Wang Yi. Dynamic budgeting for settling dram contention of co-running hard and soft real-time tasks. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 151–159, 2014.

[52] Björn Forsberg, Luca Benini, and Andrea Marongiu. Heprem: Enabling predictable gpu execution on heterogeneous soc. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 539–544, 2018.

[53] Sergio Garcia Esteban, Alejandro Serrano Cases, Jaume Abella Ferrer, Enrico Mezzetti, and Francisco Javier Cazorla Almeida. Quasi isolation qos setups to control mpsoc contention in integrated software architectures. In *35th Euromicro Conference on Real-Time Systems (ECRTS 2023): July 11-14, 2023, Vienna, Austria.*, volume 262, pages 5–1. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[54] Vıctor Garcıa, Juan Gomez-Luna, Thomas Grass, Alejandro Rico, Eduard Ayguade, and Antonio J. Pena. Evaluating the effect of last-level cache sharing on integrated gpu-cpu systems with heterogeneous applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[55] Giovani Gracioli and Antônio Augusto Fröhlich. On the design and evaluation of a real-time operating system for cache-coherent multicore architectures. *SIGOPS Oper. Syst. Rev.*, 49(2):2–16, January 2016.

[56] Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings*

in Informatics (LIPIcs), pages 27:1–27:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[57] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar)*, pages 1–10, 2012.

[58] Cuda green contexts, 2025. Accessed: 2025-12-15.

[59] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, page 245–254, New York, NY, USA, 2009. Association for Computing Machinery.

[60] N Gurushankar. Challenges in verifying complex soc designs. *J Artif Intell Mach Learn & Data Sci 2022*, 1(1):1907–1911, 2023.

[61] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *2009 30th IEEE Real-Time Systems Symposium*, pages 68–77, 2009.

[62] Mohamed Hassan. Managing dram interference in mixed criticality embedded systems. In *2019 31st International Conference on Microelectronics (ICM)*, pages 253–257, 2019.

[63] Mohamed Hassan, Anirudh M. Kaushik, and Hiren Patel. Predictable cache coherence for multi-core real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–246, 2017.

[64] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, June 2009.

[65] Isolbench benchmark suite., 2025. Accessed: 2025-02-25.

[66] Ivan Izhbirdeev, Denis Hoornaert, Weifan Chen, Alexander Zuepke, Youssef Hammad, Marco Caccamo, and Renato Mancuso. Coherence-aided memory bandwidth regulation. In *2024 IEEE Real-Time Systems Symposium (RTSS)*, pages 322–335, 2024.

[67] Shubham Kamdar and Neha Kamdar. big. little architecture: Heterogeneous multicore processing. *International Journal of Computer Applications*, 119(1):35–38, 2015.

[68] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. Managing gpu concurrency in heterogeneous architectures. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 114–126, 2014.

[69] Dae-Hwan Kim. Evaluation of the performance of gpu global memory coalescing. *Evaluation*, 4(4):1–5, 2017.

[70] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, 2014.

[71] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pages 111–122, 2004.

[72] Tomasz Kloda, Giovani Gracioli, Rohan Tabish, Reza Mirosanlou, Renato Mancuso, Rodolfo Pellizzoni, and Marco Caccamo. Lazy load scheduling for mixed-criticality applications in heterogeneous mpsocs. *ACM Transactions on Embedded Computing Systems*, 22(3), 2023.

[73] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*,

ISCA '81, page 81–87, Washington, DC, USA, 1981. IEEE Computer Society Press.

[74] Andreas Kurth, Pirmin Vogel, Andrea Marongiu, and Luca Benini. Scalable and efficient virtual memory sharing in heterogeneous socs with tlb prefetching and mmu-aware dma engine. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 292–300, 2018.

[75] Edward A. Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, 2008.

[76] Edward A. Lee. Cps foundations. In *Design Automation Conference*, pages 737–742, 2010.

[77] Hyung Gyu Lee, Naehyuck Chang, Umit Y. Ogras, and Radu 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.*, 12(3), May 2008.

[78] Jaewoo Lee and Keumseok Koh. Scheduling complex cyber-physical systems with mixed-criticality components. *Systems*, 11(6):281, 2023.

[79] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 173–189, New York, NY, USA, 2022. Association for Computing Machinery.

[80] Tamara Lugo, Santiago Lozano, Javier Fernández, and Jesus Carretero. A survey of techniques for reducing interference in real-time applications on multicore platforms. *IEEE Access*, 10:21853–21882, 2022.

[81] Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification tech-

niques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3), June 2019.

[82] Maxim Mattheeuws, Björn Forsberg, Andreas Kurth, and Luca Benini. Analyzing memory interference of fpga accelerators on multicore hosts in heterogeneous reconfigurable socs. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1152–1155, 2021.

[83] John McCalpin. Memory bandwidth and machine balance in high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, pages 19–25, 12 1995.

[84] Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1651–1657, 2018.

[85] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 146–160, 2007.

[86] Kartik Nagar and Y. N. Srikant. Fast and precise worst-case interference placement for shared cache analysis. *ACM Trans. Embed. Comput. Syst.*, 15(3), March 2016.

[87] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143, 2012.

[88] Marco Pagani, Enrico Rossi, Alessandro Biondi, Mauro Marinoni, Giuseppe Lipari, and Giorgio Buttazzo. A Bandwidth Reservation Mechanism for AXI-Based Hardware Accelerators on FPGAs. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings*

in Informatics (LIPIcs), pages 24:1–24:24, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[89] Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert, and Arno Luppold. Automated generation of time-predictable executables on multicore. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, RTNS '18, page 104–113, New York, NY, USA, 2018. Association for Computing Machinery.

[90] Hayeon Park, Jiwoong Lee, Hoyong Lee, Taekyoung Kwon, Wan Choi, Sangmi Moon, and Chang-Gun Lee. A field practical approach to memory bandwidth allocation for consolidating multi-domain automotive applications on a single soc. In *2025 IEEE 31st Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 95–107, 2025.

[91] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '06, page 67–72, New York, NY, USA, 2006. Association for Computing Machinery.

[92] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, 2011.

[93] Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, and Lui Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *2008 Real-Time Systems Symposium*, pages 221–231, 2008.

[94] Hesoc-mark benchmark suite., 2020. Accessed: 2025-12-14.

[95] Polybench benchmark suite., 2025. Accessed: 2025-02-28.

[96] Lucia Pons, Vicent Selfa, Julio Sahuquillo, Salvador Petit, and Julio Pons. Improving system turnaround time with intel cat by identifying llc critical applications. In *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27 - 31, 2018, Proceedings*, page 603–615, Berlin, Heidelberg, 2018. Springer-Verlag.

[97] Chris Porter, Chao Chen, and Santosh Pande. Compiler-assisted scheduling for multi-instance gpus. In *Proceedings of the 14th Workshop on General Purpose Processing Using GPU*, GPGPU '22, New York, NY, USA, 2022. Association for Computing Machinery.

[98] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4), January 2012.

[99] Francesco Restuccia, Alessandro Biondi, Mauro Marinoni, Giorgiomaria Cicero, and Giorgio Buttazzo. Axi hyperconnect: A predictable, hypervisor-level interconnect for hardware accelerators in fpga soc. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[100] Francesco Restuccia, Marco Pagani, Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo. Is your bus arbiter really fair? restoring fairness in axi interconnects for fpga socs. *ACM Trans. Embed. Comput. Syst.*, 18(5s), October 2019.

[101] Ahsan Saeed, Dakshina Dasari, Dirk Ziegenbein, Varun Rajasekaran, Falk Rehm, Michael Pressler, Arne Hamann, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Memory utilization-based dynamic bandwidth regulation for temporal isolation in multi-cores. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 133–145, 2022.

[102] Ahsan Saeed, Denis Hoornaert, Dakshina Dasari, Dirk Ziegenbein, Daniel Mueller-Gritschneder, Ulf Schlichtmann, Andreas Gerstlauer, and Renato Mancuso. Memory Latency Distribution-Driven Regulation for Temporal Isolation in MPSoCs. In Alessandro V. Papadopoulos, editor, *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, volume 262 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:23, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[103] Urvij Saroliya, Eishi Arima, Dai Liu, and Martin Schulz. Hierarchical resource partitioning on modern gpus: A reinforcement learning approach. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 185–196, 2023.

[104] Eric Seals, Michael Bechtel, and Heechul Yun. Bandwatch: A system-wide memory bandwidth regulation system for heterogeneous multicore. In *2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 38–46, 2023.

[105] Alejandro Serrano-Cases, Juan M. Reina, Jaume Abella, Enrico Mezzetti, and Francisco J. Cazorla. Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:26, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[106] Alan Jay Smith. Multiprocessor memory organization and memory interference. *Commun. ACM*, 20(10):754–761, October 1977.

[107] Parul Sohal, Michael Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. A closer look at intel resource director technology (rdt). In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, RTNS ’22, page 127–139, New York, NY, USA, 2022. Association for Computing Machinery.

[108] Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. E-warp: A system-wide framework for memory bandwidth profiling and management. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 345–357, 2020.

[109] Andrea Stevanato, Matteo Zini, Alessandro Biondi, Bruno Morelli, and Alessandro Biasci. Learning memory-contention timing models with automated platform profiling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3816–3827, 2024.

[110] Noriaki Suzuki, Hyoseung Kim, Dionisio de Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 685–692, 2013.

[111] Michael B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 1131–1136, New York, NY, USA, 2012. Association for Computing Machinery.

[112] Giacomo Valente, Gianluca Brilli, Tania Di Mascio, Alessandro Capotondi, Paolo Burgio, Paolo Valente, and Andrea Marongiu. Fine-grained qos control via tightly-coupled bandwidth monitoring and regulation for fpga-based heterogeneous socs. *IEEE Transactions on Parallel and Distributed Systems*, 36(2):326–340, 2025.

[113] Giacomo Valente, Tiziana Fanni, Carlo Sau, Tania Di Mascio, Luigi Pomante, and Francesca Palumbo. A composable monitoring system for heterogeneous embedded platforms. *ACM Trans. Embed. Comput. Syst.*, 20(5), July 2021.

[114] Prathap Kumar Valsan and Heechul Yun. Medusa: A predictable and high-performance dram controller for multicore based embedded systems. In *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*, pages 86–93, 2015.

[115] Prathap Kumar Valsan, Heechul Yun, and Farzad 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)*, pages 1–12, 2016.

[116] Hao Wen and Zhang Wei. Interference evaluation in cpu-gpu heterogeneous computing. In *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017.

[117] Jun Yan and Wei Zhang. Wcet analysis for multi-core processors with shared l2 instruction caches. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89, 2008.

[118] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. Coloris: a dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 381–392, New York, NY, USA, 2014. Association for Computing Machinery.

[119] Huixuan Yi, Yuanhai Zhang, Zhiyang Lin, Haoran Chen, Yiyang Gao, Xiaotian Dai, and Shuai Zhao. A cache-aware dag scheduling method on multicores: Exploiting node affinity and deferred executions. *J. Syst. Archit.*, 161(C), April 2025.

[120] Heechul Yun. Understanding and mitigating hardware interference channels on heterogeneous multicore. In *2024 IEEE 3rd Real-Time and Intelligent Edge Computing Workshop (RAGE)*, pages 1–3, 2024.

[121] Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms. *IEEE Transactions on Computers*, 66(7):1247–1252, 2017.

[122] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, 2014.

[123] Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 184–195, 2015.

[124] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

[125] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2016.

[126] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 89–102, New York, NY, USA, 2009. Association for Computing Machinery.

[127] Yanqi Zhou and David Wentzlaff. Mitts: memory inter-arrival time traffic shaping. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 532–544. IEEE Press, 2016.

[128] Matteo Zini, Daniel Casini, and Alessandro Biondi. Analyzing arm's mpam from the perspective of time predictability. *IEEE Transactions on Computers*, 72(1):168–182, 2023.

[129] Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. Profiling and controlling i/o-related memory contention in cots heterogeneous platforms. *Software: Practice and Experience*, 52(5):1095–1113, 2022.

[130] Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. Mempol: Policing core memory bandwidth from

outside of the cores. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–248, 2023.