Observe and Regulate Memory Interference on MPSoC: a Practical Approach

Andrea Bastoni*†, Alexander Zuepke*†, Marco Solieri†
Technical University of Munich*, Minerva Systems†
andrea.bastoni@tum.de, alex.zuepke@tum.de, marco.solieri@minervasys.tech

***Abstract*—The increasing deployment of AI-enabled cyber-physical applications in industrial automation, robotics, automotive, and avionics requires high computational power with strict timing guarantees. Heterogeneous multiprocessor systems-on-chip (MPSoCs) have emerged as a solution, integrating application cores, real-time cores, and accelerators to balance performance and energy efficiency. However, shared memory hierarchies adopted by MPSoC introduce contention, potentially jeopardizing the predictability of real-time tasks. To mitigate these memory-related issues, both hardware-based Quality-of-Service (QoS) and software-based memory monitoring and regulation strategies are available. However, in dependable and real-time contexts, QoS mechanisms are not always effective and, in general, configuring both hardware- and software-based regulation optimally remains challenging. This paper presents insights into *MemPol*, a recent software-based memory regulation approach, and its integration into Minerva Systems *Architect*, an industrial tool for managing memory contention in MPSoCs. Through a real-world use case, we demonstrate how systematic memory monitoring and regulation can preserve timing guarantees while maximizing system utilization.**

I. Introduction

Modern applications in the industrial automation, robotics, automotive, and avionics domains are increasingly deployed "on the edge" and require high computational power with limited power consumption. Many of these cyber-physical applications perform critical operations and must satisfy stringent timing requirements to prevent *e.g.*, injuries or business-related damages. To meet the demand of these applications, traditional microcontroller-based architectures have transitioned to high-performance, low-power heterogeneous multiprocessor systems-on-chip (MPSoCs). Such systems integrate CPUs, GPUs, AI accelerators, and real-time cores, delivering high computational power with efficient energy usage.

For example, in the automotive domain, modern onboard ADAS (Advanced Driver Assistance Systems) utilize CPUs, GPUs, and AI accelerators to process data from multiple cameras, LiDAR, and radar. These systems enable real-time object detection and trigger emergency braking procedures or assist with lane-keeping. Similar applications exist in robotics, where high-performance real-time sensor fusion and AI-based decision-making are essential for operating autonomous robots.

To maximize efficiency and performance, MPSoCs (*e.g.*, [1], [2]) rely on shared memory hierarchies, introducing potential contention for critical applications that must compete at the cache, interconnect, and memory-controller levels. Memory contention poses a significant challenge for real-time tasks, as their timing guarantees may be affected by best-effort tasks with aggressive memory workloads. Additionally, interference can compromise separation properties, which are essential in many safety and security certification processes.

An example of problematic contention at memory level can be seen in Fig. 1, which shows the impact of different types of memory interference on the execution time of a hypervisor-isolated real-time application running on an AMD Ultrascale+ ZCU102 MPSoC [1]. As visible in the figure, the application is sensitive to thrashing in the L2 cache and to interference in the memory controller (*WC Write* and *Linear Write*), but is relatively unaffected by cache *flush* or *clean* operations on the shared L2 cache.

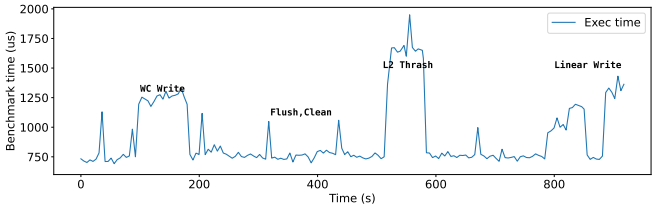

Fig. 1. Tracing of execution time (us) variability of a benchmark application under different types of memory interference using Minerva Systems *Architect* [3] on ZCU102 [1].

To address these issues, hardware vendors provide quality-of-service (QoS) mechanisms such as Intel RDT, Arm MPAM, and Arm QoS-400 [4]–[6], allowing monitoring and regulation of memory bandwidth consumption. However, as shown in recent studies (*e.g.*, [7], [8]), these mechanisms are difficult to correctly configure, might not provide the expected level of protection, and are not always available on embedded MPSoCs.

In the literature, several strategies have been proposed to leverage the monitoring capabilities of MPSoCs' Performance Monitoring Units (PMUs) and implement software-based regulation, which has proven effective in mitigating certain types of memory interference [9]–[11]. Nonetheless, due to the numerous control parameters and their complex interactions, finding the optimal configurations to achieve both isolation and performance remains a major challenge.

This paper presents key insights into *MemPol*, a recent software-based memory monitoring and regulation strategy [11] and its transition to Minerva Systems *Architect* [3],

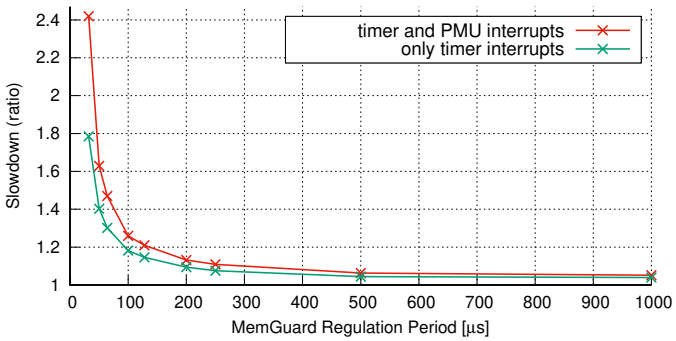

Fig. 2. Adapted from [11]. Impact (slowdown) of *MemGuard*'s timer and regulation overheads on a memory-intensive application as a function of the replenishment period. Results are in line with other works [9], [15].

an industrial tool designed to support the development of complex cyber-physical applications on MPSoCs. *Architect* enhances traditional processes with a developer-driven, iterative approach to identifying, analyzing, and resolving potential memory issues, effectively helping integrators achieve higher system utilization, while ensuring predictable performance.

To illustrate the practicality of this approach, we present a real-world example, demonstrating how memory monitoring and regulation can be effective in preserving timing guarantees while delivering high performance.

II. Approaches to Memory Regulation

Several software- and hardware-based techniques have been proposed to mitigate uncontrolled memory interference.

MemGuard [9] first proposed a software approach that levarage performance counters (PMC) to regulate memory bandwidth at the OS or hypervisor level. Methods following *MemGuard*-based strategy [12]–[14] regulate the maximum number of memory transactions that cores can perform over a defined period of time. Cores are assigned a memory budget that is periodically replenished and that is consumed when cores perform memory transactions. Cores idle when the budget is depleted. *MemGuard* approaches rely on PMCs to monitor memory bandwidth and require a mechanism to deliver replenishment interrupts and idle the cores. Although short periods are desirable to enforce fine-grained regulation, the excessive overheads caused by frequent interrupts make high-frequency *MemGuard* regulation non viable [15].

As an example, Fig. 2 reports the overheads of timer-replenishment and regulation interrupts in the *MemGuard* setup used in [11]. The figure shows the slowdown of a memory-intensive application as function of the replenishment period. The budget is measured as the number of L2 cache refills. Fig. 2 separately shows the impact of timer *and* regulation (PMU) interrupt, and timer interrupts only. As shown, for short regulation periods ($32\ \mu s$), *MemGuard* is affected by extremely high overhead—up to 2.4 slowdown ratio.

The recently proposed *MemPol* [11] software-based regulation targets some of *MemGuard* limitations and proposes a low-overhead memory monitoring and regulation mechanism that operates "from outside" of the main cores at high frequency. Sec. III presents *MemPol* in greater details.

Modern MPSoCs may provide additional QoS and monitoring features (*e.g.*, [16]) that can be used alternatively to PMCs. Although effective, such mechanisms monitor at platform interconnect level, making it difficult to attribute memory traffic to the generating cores. Furthermore, their availability is limited to subsets of COTS boards. Several studies [17]–[20] have explored such primitives to realize bandwidth regulation.

Comprehensive monitor and regulation strategies such as Intel RDT, Arm MPAM [4], [6] might be available on selected MPSoCs, but they might fail to provide the expected level of protection [8]. Additionally, Arm MPAM defines all control interfaces as "optional", requiring a careful analysis of the knobs available in actual implementations.

Hardware-based approaches to memory bandwidth regulation (*e.g.*, [21]–[24]) propose the development of custom hardware or FPGA logic to realize fine-granularity regulation at hardware level.

Orthogonal to memory bandwidth regulation, to control interference at cache and DRAM level, techniques such as [10], [25]–[27] have been proposed. Hardware support to cache-partitioning is also available on recent Arm-based boards [28].

III. Background on MemPol

The software-based regulation recently proposed by *MemPol* [11] aims to overcome limitations of *MemGuard*-based approaches. Specifically, *MemPol*'s design targets the high overheads of high-frequency interrupt-based regulation, and enables flexible regulation policies based on the simultaneous monitoring of multiple platform dimensions.

The high-frequency, low-overhead design is achieved operating from the outside of the target cores. *MemPol* monitors the last-level cache (LLC) activity by polling the cores' PMCs and uses a core-independent interface (*e.g.* the *CoreSight* debugging interface) to halt cores when they exceed their given memory budget.

Furthermore, *MemPol* realizes a multi-dimensional regulation based on the combined contribution of multiple PMCs. This overcomes one limitation of *MemGuard*-based approaches that can only monitor one PMU dimension at a time. Although the regulation in [11] is based on the accumulated read and write activities of a core, depending on the platform and on the availability of PMC, different types of regulation can be enacted.

Compared to *MemGuard*, *MemPol*'s regulation logic is performed every polling period P using an on-off controller policy that can idle cores for short time intervals (at least P). Due to polling, the *MemPol* controller can observe the potentially high numbers of transactions generated by monitored cores only during the next polling interval. This overshooting-effect is countered using a short polling period P in the microsecond range. Instead, due to overheads, *MemGuard* approaches can only operate in the millisecond range.

The low-overhead, high-resolution capabilities of *MemPol* can be used to realize both local per-core regulation policies

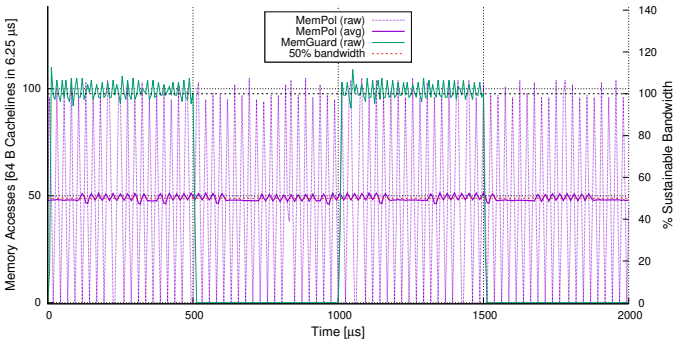

Fig. 3. Adapted from [11]. Comparison of the regulation behavior of *MemPol* (polling at 6.25 µs) and *MemGuard* (regulation period 1 ms) on AMD Ultrascale+ ZCU102 regulating a worst-case memory reader at 50% memory bandwidth. For *MemPol*, the average over 200 µs is also shown for better visualization of its resulting regulation.

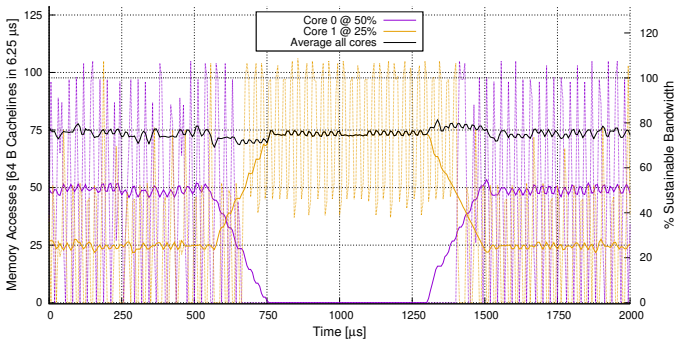

Fig. 4. Adapted from [11]. *MemPol* bandwidth redistribution: Core c_0 is regulated at 50% bandwidth and alternates memory access and idle phases every 750 µs. Core c_1 is regulated at 25% bandwidth and accesses memory all the time. Both cores perform worst-case reading. The global regulator is enabled and redistributes unused bandwidth from c_0 to c_1 while c_0 is idle, but keeps the overall bandwidth at 75%, which the sum of both cores' configured bandwidth. Solid lines average over 200 µs. Polling at 6.25 µs.

(similar to *MemGuard*), as well as global regulation policies that can (dynamically) redistribute unused bandwidth not reclaimed by cores, while still keeping the overall bandwidth below a global memory budget.

The implementation of *MemPol* relies on the PMU register interface to monitor the PMCs of a core and simulates the behavior of a debugger to stall and restart cores. The control logic is typically deployed on one of the (small, real-time) cores available on MPSoCs. For example, in [11], one Cortex-R5 or one Cortex-M4/7 are used, depending on the MPSoC. However, the control logic can also be deployed on large application cores. We invite the interested reader to refer to [11] for more details. An example of *MemPol*'s design—within Minerva Systems *Architect* [3] on a ZCU102—is shown in Fig. 6. Here, *MemPol* is deployed on one real-time Cortex-R5 core and operates only from its TCM memory to avoid interfering with the monitored and regulated Cortex-A53 cores.

Fig. 3 presents a comparison of the fine-grained regulation preformed by *MemPol* and the coarse-grained one by *MemGuard*. In the example, both mechanisms achieve the same regulation results over a longer time span, but *MemPol* regulates at a higher frequency (6.25 µs) than *MemGuard* (1 ms).

Instead, Fig. 4 shows the redistribution of unused memory bandwidth that can be enacted by *MemPol*'s global regulator. In the figure, core c_0 (regulated at 50%) alternates between memory access and idle phases, while core c_1 (regulated at 25%) always performs memory accesses. Using the global regulator, c_1 is allowed to use any remaining bandwidth up to the global configured limit of 75%. The overshooting due to *MemPol*'s polling behavior is visible in Fig. 4 when c_0 returns from being idle, as the local regulator for c_0 lets the core consume the bandwidth up to its budget.

IV. Industry Transfer

Research-driven approaches such as *MemPol* [11] and *MemGuard* [9] are effective memory regulation and monitoring strategies for MPSoCs. However, their practical adoption in industrial embedded and cyber-physical systems remains limited. In fact, these methods require significant expertise in MPSoC architectures for proper implementation, deployment, and result analysis—expertise that is typically found in academic research rather than industrial development teams.

In industrial contexts, developers and system integrators are domain experts focused on application-specific functionalities rather than low-level platform behaviors. Furthermore, applications are typically tested in isolation, and platform-related performance bottlenecks or potential memory interference issues are often discovered late in the integration phase. At this stage, resolving such issues can be highly disruptive, leading to delays, increased costs, and potential compromises in system functionality. In some cases, unresolved performance problems may necessitate costly hardware upgrades or force developers to scale back system features.

To bridge the gap between academic research and industrial application, memory regulation strategies like *MemPol* must be integrated into a broader, structured framework. This framework should seamlessly align with established industrial methodologies, supporting iterative, developer-driven workflows and potentially certified processes.

Moving toward this goal, we integrated the memory monitoring and regulation provided by *MemPol* into Minerva Systems' *Architect* [3] framework. The objective is to create a tool to help developers and integrators address the challenges of safe and secure integration on complex MPSoCs and optimize application performance.

A. Architect *Features*

Memory Tracing and Inspection. *Architect* enables developers to visualize live memory traffic and memory interference generated by concurrently executing applications. A variant of *MemPol* is used to collect memory traces without interfering with the applications. Using the GUI, memory-access patterns and memory interference can be visually correlated with applications running on each core and with additional information coming from standard performance monitoring (*e.g.*, perf on Linux). Problematic and unexpected cross-core

Fig. 5. *Architect* GUI trace example on a quad-core ZCU102 MPSoC [1]. The selected process (`target`) is marked in blue in the upper part of the figure (second CPU). Orange tasks are potential interference. The lower part of the picture visualize memory accesses. The pop-up control panel shows some of the statistics and tunables that developers can access during an interactive session (*e.g.*, CPU affinity, process tree filtering).

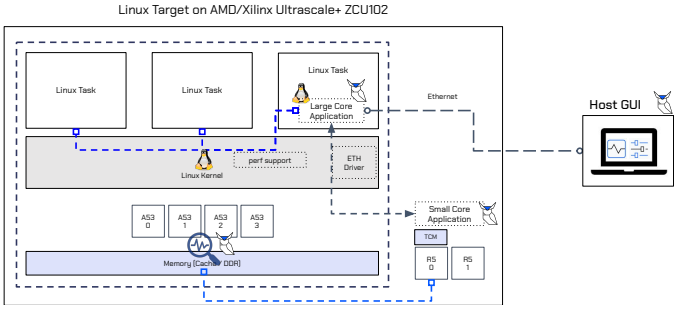

Fig. 6. *Architect* deployment for a Linux target system on a ZCU102. The "owls" identify key components of the architecture: host GUI, monitoring and regulation logic between Cortex-A53 cores and one of the Cortex-R5 core, `target` applications collecting performance information from Linux task and memory-related information.

memory interference can thus be quickly identified. *Architect* offers detectors for tasks periodicity and interrupt tracing, enabling faster identification of recurring patterns.

Interference Generator. Generating memory stress tests require significant knowledge of MPSoC architectures. *Architect* simplifies this task with pre-configured memory stress-test scenarios for multiple MPSoCs. Scenarios can be controlled interactively via the GUI, or in an automated way. Multiple complex memory stress patterns are available by default and the execution of each interference generator can be customized by selecting the priority and the executing core.

Interference Control Capabilities. The GUI provides interactive access to regulation settings for different (board-dependent) variants of memory bandwidth regulation. The settings directly operate on the (appropriately tuned) memory-regulation capabilities of *MemPol*. The impact of different regulation levels can be observed live, and adjusted according to the type of workload and interference. Once identified, the best strategies for memory regulation and interference mitigations can be integrated with the running operating system and/or hypervisor. For example, pre-configured integrations for multiple flavors of Linux and the Jailhouse hypervisor have been developed. Fig. 5 illustrates some of the visualization and control options of *Architect*.

Interference Testing Automation. *Architect* enables monitoring of regressions during incremental integration and progressively complex testing activities. The interactive configurations that can be activated from the GUI can be replicated and activated automatically during long-running testing activities to facilitate regression testing and support certification evidence. A simple example of the potential of the Python-based API is shown in Listing 1. The API uses coroutines to asynchronously coordinate the execution of actions on the target platform. The example shows a simplified version of the tests discussed in Sec. V. Lines 3 and 17 start and stop the memory bandwidth regulation, while lines 7 and 9 update the memory budget and start interference on the selected cores.

B. Architecture and Supported Platforms

The framework adopts a host / target split architecture that only requires an Ethernet connection between the target SoC and the host of the developer / integrator. Analysis and control activities are triggered via a responsive GUI that visualizes run-time memory and application information from the target with high resolution. The architecture is illustrated in Fig. 6 for a Linux deployment on an AMD/Xilinx Ultrascale+ ZCU102 MPSoC [1]. The MPSoC features four Arm Cortex-A53 whose

Listing 1. Example of *Architect* API Usage

```
async def test(target: Target):
        # start regulation on target
        await target.membw_ctrl.enable()
        for budget in range(600, 1600, 100):
                for core in range (0, 2, 1):
                        # update the budget for the current round
                        await target.membw_ctrl.adjust(core, budget)
                        # start interference
                        await target.interference.hog_start(core, 1, 1)

                # run for 40 seconds
                await asyncio.sleep(40)

                for core in range (0, 2, 1):
                        await target.interference.hog_stop(core)
        # stop the regulation
        await target.membw_ctrl.disable()
```

memory accesses are monitored from one of the Cortex-R5 available on the board using a variant of *MemPol*. On Linux, *Architect* leverages the perf infrastructure to collect task-related information such as context switches and execution time, and implements a target-side application (`target`) that collects both memory and perf tracing data. `target` implements the communication endpoint with the system's GUI that retrieves the data, visualizes it at high resolution (60 FPS) and also stores it for offline detailed analysis.

Compared to the platforms supported by *MemPol*, *Architect* support has been extended to a wider range of Arm v8 families, including AMD Ultrascale+ (Kria KV26, KR26, ZCU102, etc.), NXP (i.MX8 and S32G), and Texas Instrument (AM62x, AM67x). On the target side, *Architect* supports bare-metal (no OS) environments, Linux, and SYSGO PikeOS [29]. On the host side, the GUI is available for Linux.

V. Example Use Case

We present the application of *Architect* in the context of a realistic industrial use case derived from a customer's deployed system. The analyzed system is implemented on an AMD UltraScale+ ZCU102 MPSoC [1], featuring four ARM Cortex-A53 cores sharing a 1 MiB L2 last-level cache (LLC), a unified memory hierarchy and an FPGA programmable logic area. The workload is distributed across the four cores following a partitioned scheduling approach. Specifically, one core is exclusively responsible for handling communication with FPGA IP blocks controlling a critical industrial appliance. Tasks assigned to this core have strict real-time constraints, requiring predictable and low-latency execution loops to guarantee the dependable operation of the appliance. Any deviation from the required timing behavior could result in significant operational disruptions and business impacts, such as interruptions of production lines.

During integration tests performed under increased workloads on the three supposedly independent cores, the customer observed deadline misses for the latency-sensitive tasks. Despite testing, traditional debugging methods proved insufficient to precisely identify the root cause of the timing degradation. By leveraging *Architect*, we monitored and analyzed memory contention and cache interference scenarios in detail. In this paper, the customer's latency-sensitive application is modeled using a representative benchmark application.

In a preliminary analysis phase, we used *Architect*'s GUI to systematically evaluate the application's sensitivity to different sources of memory interference. Through the GUI, we interactively introduced several interference scenarios, executing on CPUs separate from the latency-sensitive tasks. The system's execution was extensively traced, capturing detailed performance data from the representative benchmark while the interference workloads was running concurrently on the neighboring cores.

Fig. 1 presents a representative sample of the observed performance degradation due to different interference types. At around 100 seconds, we introduce a worst-case memory write pattern, resulting in observable but limited performance degradation. Between seconds 100 and 500, cache flush and cache clean operations are performed on the shared L2 cache; the benchmark remained relatively unaffected, confirming limited sensitivity to these operations. However, a significant performance degradation can be seen at approximately second 500 when thrashing in the shared L2 cache is introduced. From second 800 onwards, we execute linear memory write interference, simulating high-bandwidth workloads from competing tasks. Under this condition, the benchmark shows severe degradation and unstable execution behavior. Specifically, around second 1000, the benchmark spikes and enters a protection mode emulating a system safety "halt" measure to prevent potential operational hazards (as such, the spikes in benchmark behavior are not shown in the figure).

Given the impact of L2 cache thrashing and linear memory write interference observed during the interactive analysis, automated tests were designed to evaluate whether memory bandwidth regulation could effectively mitigate these interference scenarios. The goal was to identify optimal memory

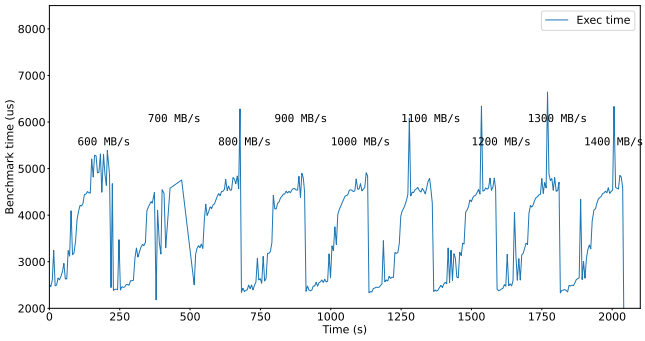

Fig. 7. Execution time of the latency-sensitive benchmark (y-axis) subject to L2 thrashing interference under progressive bandwidth regulation (from 600 to 1400 MB/s). Note that the time intervals differ from Fig. 8.

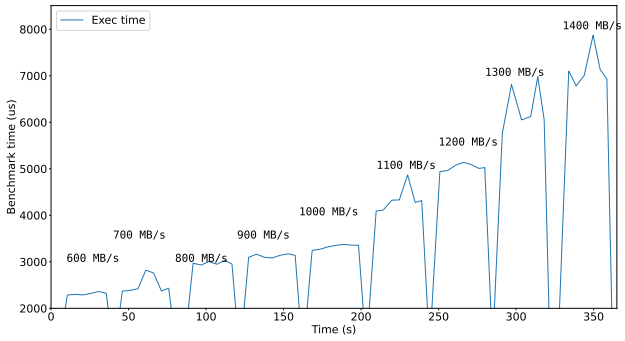

Fig. 8. Execution time of the latency-sensitive benchmark (y-axis) subject to linear write memory interference under progressive bandwidth regulation (from 600 to 1400 MB/s). Note that the time intervals differ from Fig. 7.

bandwidth regulation levels that prevent timing violations in latency-sensitive tasks while maintaining high utilization of system resources. Leveraging the *Architect* API, we implemented a series of automated experiments that systematically apply memory bandwidth regulation under controlled interference conditions. Specifically, the system was stressed via either L2 thrashing or linear memory write interference patterns on cores distinct from the latency-sensitive application. During each experiment, the allowed memory bandwidth on the interfering cores was gradually increased in periodic increments, monitoring the timing behavior of the latency-sensitive benchmark. This iterative approach continued until the latency-sensitive task approached or exceeded predefined safety bounds.

Figures 7 and 8 illustrate the impact of memory bandwidth regulation on L2 thrashing and linear memory write workloads, respectively. Each interval has variable duration (150 seconds in Fig. 7 and 30 seconds in Fig. 8) and presents the execution duration (y-axis) of the latency-sensitive benchmark under progressively increasing memory bandwidth constraints applied to the interfering cores. Note that the benchmark dimension monitored in these tests differs from the dimension monitored in Fig. 1 and that the y-axis therefore differs.

Bandwidth limitations were increased from 600 MB/s to 1400 MB/s, with the goal of maximizing available bandwidth for the interfering cores without violating timing constraints on the latency-sensitive tasks. The tests revealed that a regulation threshold of 1200 MB/s provides the most suitable balance between these two dimensions. As expected, the results also indicate that memory bandwidth regulation is particularly effective against linear memory accesses interference patterns. Instead, the L2 thrashing interference scenario showed comparatively limited sensitivity to bandwidth regulation, as indicated by persistent latency spikes in Fig. 7.

The results show that additional measures are required to improve predictability under heavy cache contention conditions. Therefore, in further iteration of the project, we studied the integration of cache partitioning techniques into the system, while monitoring latency-sensitive application benchmark when increasing both system utilization and cache pressure.

VI. Conclusion

The increasing computational demands and stringent timing requirements of modern AI-enabled applications have led to the adoption of high-performance, low-power heterogeneous MPSoCs. However, these systems' reliance on shared memory hierarchies can introduce significant resource contention and achieving predictable real-time performance for dependable, critical applications becomes a challenge.

This paper presented key insights into *MemPol* [11], a recent software-based memory monitoring and regulation strategy and its integration into Minerva Systems *Architect* [3], an industrial tool designed to optimize the execution of critical applications on MPSoC, and in particular to facilitate the analysis and mitigation of memory interference

The high-frequency, low-overhead monitoring realized by *MemPol* is integrated in *Architect* and complemented with features such as live-tracing, interference generation and control, and testing automation that helps developers to address the challenges of safe and secure integrations of complex applications on MPSoCs.

Our real-world case study demonstrated that the combination of these features effectively ensures timing guarantees and high performance, even under demanding interference conditions such as L2 cache thrashing and linear memory workloads. Additionally, our analysis can indicate problematic interference that cannot be mitigated by memory bandwidth regulation (*e.g.*, that requires cache partitioning strategies) and provides measurable metrics to also quantify the improvements deriving from the implementation of such techniques.

Acknowledgments

A. Bastoni and A. Zuepke were supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.

References

[1] AMD, *Zynq UltraScale+ Device TRM*, https://docs.amd.com/r/en-US/ug1085-zynq-ultrascale-trm/.

[2] NVIDIA, *NVIDIA Jetson AGX Orin*, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/.

[3] Minerva Systems, *Architect*, https://minervasys.tech/architect.

[4] Intel, *Resource Director Technology*, https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html.

[5] Arm, *ARM CoreLink QoS-400*, https://developer.arm.com/docs/dsu0026/.

[6] Arm, *Arm Memory System Resource Partitioning and Monitoring (MPAM)*, https://developer.arm.com/documentation/ihi0099/.

[7] M. Bechtel and H. Yun, "Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention," in *RTAS 2019*.

[8] P. Sohal *et al.*, "A Closer Look at Intel Resource Director Technology (RDT)," in *RTNS 2022*.

[9] H. Yun *et al.*, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *RTAS 2013*.

[10] T. Kloda *et al.*, "Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems," in *RTAS 2019*.

[11] A. Zuepke *et al.*, "MemPol: Polling-based Microsecond-scale Per-core Memory Bandwidth Regulation," *Real-Time Systems*, vol. 60, 2024.

[12] P. Modica *et al.*, "Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms," in *2018 IEEE International Conference on Industrial Technology (ICIT)*, 2018, pp. 1651–1657.

[13] N. Dagieu, A. Spyridakis, and D. Raho, "Memguard: A memory bandwith management in mixed criticality virtualized systems memguard KVM scheduling," in *10th Int. Conf. on Mobile Ubiquitous Comput., Syst., Services and Technologies (UBICOMM)*, 2016.

[14] J. Martins *et al.*, "Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems," in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, M. Bertogna and F. Terraneo, Eds., ser. OpenAccess Series in Informatics (OASIcs), vol. 77, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, 3:1–3:14.

[15] A. Saeed *et al.*, "Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores," in *2022 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 133–145.

[16] Arm, *Quality of Service in ARM Systems: An Overview*, https://community.arm.com/arm-community-blogs/b/soc-design-and-simulation-blog/ posts/quality-of-service-in-arm-systems-an-overview Accessed: 2025-03-09.

[17] P. Sohal *et al.*, "E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020.

[18] A. Serrano-Cases *et al.*, "Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, B. B. Brandenburg, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 196, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 3:1–3:26.

[19] P. Houdek, M. Sojka, and Z. Hanzálek, "Towards predictable execution model on ARM-based heterogeneous platforms," in *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*, 2017, pp. 1297–1302.

[20] M. Zini *et al.*, "Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms," *Software: Practice and Experience*, vol. 52, no. 5, pp. 1095–1113, 2022.

[21] Y. Zhou and D. Wentzlaff, "MITTS: Memory Inter-Arrival Time Traffic Shaping," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16, Seoul, Republic of Korea: IEEE Press, 2016, pp. 532–544.

[22] F. Farshchi, Q. Huang, and H. Yun, "BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 364–375.

[23] J. Cardona *et al.*, "Maximum-contention control unit (MCCU): resource access count and contention time enforcement," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, J. Teich and F. Fummi, Eds., IEEE, 2019, pp. 710–715.

[24] N.-J. Wessman *et al.*, "De-RISC: The first RISC-V space-grade platform for safety-critical systems," in *2021 IEEE Space Computing Conference (SCC)*, IEEE, 2021, pp. 17–26.

[25] R. Mancuso *et al.*, "Real-time cache management framework for multi-core architectures," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 45–54.

[26] X. Project, *Xen 4.20 Support with Cache-Coloring*, https://wiki.xenproject.org/wiki/Xen_Project_4.20_Feature_List Accessed: 2025-03-09.

[27] H. Yun *et al.*, "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)*, 2014, pp. 155–166.

[28] Arm, *Arm DynamIQ Shared Unit Technical Reference Manual*, https://developer.arm.com/documentation/100453/ Accessed: 2025-03-09.

[29] SYSGO GmbH, *PikeOS Hypervisor*, https://www.sysgo.com.