Octopus: gem5-Integrated Rapid Prototyping for Resource Contention Measurement and Control

Yuying Lai*, Guotong Miao*, Shorouk Abdelhalim*, Mohamed Hossam*,
Yazi Chen†, Rodolfo Pellizzoni†, Mohamed Hassan*
*{laiy24,miaog,abdels28,mohamed.hossam,mohamed.hassan}@mcmaster.ca, †{yazi.chen, rpellizz}@uwaterloo.ca
*McMaster University, Hamilton, Ontario, Canada, †University of Waterloo, Waterloo, Ontario, Canada

***Abstract*—To address the resource contention issue in high-performance real-time Systems-on-a-chip (SoCs), the research community has proposed many different designs for components in the memory hierarchy, including memory controllers, interconnections and coherent caches. Most such proposals have been evaluated using custom standalone simulators. This makes it difficult to compare alternative designs. Furthermore, our experience is that implementing the simulator from scratch requires significant time, slowing down the evaluation of new ideas. Following the ECRTS Arm Industrial Challenge, we propose Octopus, a gem5-integrated architectural simulator that serves as a prototyping platform to measure and control the impact of resource contention. Octopus is a highly modular simulation model for the whole memory hierarchy, including caches, interconnect and main memory, which is specifically designed to simplify the testing and implementation of predictable arbitration schemes for mixed-critical systems. Unlike gem5's Ruby memory model, Octopus is designed with real-time systems as a first-class citizen. We further integrate our model with the ARM Adaptive Traffic Profile (ATP) gem5 engine to simulate the injection of various traffic profiles for competing masters in the systems. As an example of the potentialities offered by our framework, we implemented and tested our GRROF hardware management scheme (published in the main conference), requiring approximately 150–200 lines of code and less than one week of development effort. GRROF provides much tightened end-to-end latency bounds for memory requests through coordination of multiple resource arbiters. Our evaluation of the AR HUD case study from the Industrial Challenge demonstrates over 20× reduction in end-to-end observed memory latency on a modern out-of-order quad-core platform, while maintaining near high-performance average throughput.**

I. Introduction

Real-time embedded systems are increasingly relying on multicore Systems-on-a-Chip (SoCs) platforms [1] to satisfy ever-growing performance demands while meeting SWaP constraints. Many such systems are mixed-critical, comprising safety- and timing-critical applications alongside low-criticality and best-effort software components. While strong functional isolation can be provided e.g. through embedded hypervisors [2], [3], guaranteeing timing isolation to applications with different criticalities is much harder: cores contend for access to shared resources in the memory hierarchy such as caches, interconnections and main memory, causing mutual timing interference. This is made even more problematic by the prevalence of heterogeneous SoCs, since accelerators such as GPUs and DPUs are harder to control from the OS perspective compared to general purpose CPU cores. For this reason, we argue that it is imperative for hardware platforms to provide suitable monitoring mechanisms and control knobs to allow timing isolation in the memory hierarchy among contending heterogeneous processors. In this direction, the research community has proposed many different designs for components in the memory hierarchy, ranging from DRAM memory controllers [4]–[15], to cache controllers [16], to bus and coherence designs [17]–[23]. Industry has similarly moved towards architectural solutions for monitoring and partitioning of resource usage, including Arm MPAM [24], Intel RDT [25], and AMD Platform QoS [26], although with a larger focus on the data center rather than the embedded market.

To further understand the problem when dealing with complex embedded applications, devise new analysis methodologies, and propose strategies for resource isolation and contention monitoring, Arm introduced an Industrial Challenge in ECRTS22 [27]. The Industrial Challenge describes an automotive case study that realizes an Augmented Reality Head-Up Display (AR HUD) application. It comprises two software components, a Simultaneous Localization and Mapping (SLAM) function comprising multiple CPU tasks running on a homogeneous Arm multicore and a DNN-based pose estimation function that runs on a GPU. To execute the application, the Industrial Challenge proposes two platforms: a virtual platform based on the gem5 simulation framework [28], and the NVIDIA Jetson Xavier NX board as a physical platform. Because the CPU and GPU share main memory, traffic generated by bandwidth-intensive GPU kernels can significantly affect the more latency-sensitive CPU tasks; when running the application on a Jetson Nano board, the authors of [29] found that running the pose estimation function on the integrated GPU resulted in a 27X increase in the trajectory error of the SLAM function. In addition, in a mixed-criticality system, high-criticality real-time tasks such as the SLAM could further suffer contention from lower-criticality tasks executing on different cores or other accelerators (e.g. DPU). To simulate such aggressor workloads on the virtual platforms, the Industrial Challenge provides a set of AMBA Adaptive Traffic Profiles (ATP) [30]. Each profile provides information on the type of memory requests issued by a hardware component and their throughput. An ATP Engine [31] can be used to generate requests according to a specified profile during simulation. On the physical platform, [29] used a set of synthetic tasks running on a separate core to further stress the system, resulting in the

complete failure of the SLAM function.

In this paper, we are interested in prototyping resource contention measurement and control mechanisms at the architectural level. For this reason, we consider a simulation environment, similar to the virtual platform introduced in the Industrial Challenge. While, as discussed above, the community has provided many designs for predictable hardware components in the memory hierarchy, most such proposals have been evaluated on custom simulators. The lack of a common infrastructure makes it difficult to compare alternative proposals, and cumbersome to develop and test new ideas. While gem5 provides the Ruby memory model [32], which simulates caches and interconnections, most surveyed papers chose not to use it. Based on our experience, this is mainly due to the challenge in making substantial modifications to the model, especially when it comes to the detailed implementation of coherence protocols. To address this issue, we propose Octopus [33], a cycle-accurate simulator for the cache hierarchy. Octopus is a highly modular and flexible simulator model, designed with real-time systems as a first-class citizen. It implements detailed modeling of cache coherence states, and is easily extensible to support alternate coherence protocols, arbitration schemes, and cache hierarchies. Octopus implements monitoring functionalities especially tailored for time-critical systems, e.g., through latency logging, and supports many of the proposed predictable cache coherent solutions (see Section III).

To show how Octopus enables the exploration of novel architectural solutions to resource contention in the context of the aforementioned Industrial Challenge, we employ the Global Round-Robin Oldest-First (GRROF) arbitration mechanism [34], our recently proposed architecture for coordinated arbitration across the memory hierarchy. GRROF allows the derivation of end-to-end latency bounds for caches, interconnections and main memory. It implements a system-wide arbitration scheme that coordinates across individual resource arbiters, preserving priorities for requests of a same core, while still allowing servicing ready requests out of order. This prevents intra-core priority inversion, which can lead to high latency spikes, while preserving throughput in the average case. In addition, by enforcing a global round-robin order across cores, GRROF enables tighter end-to-end real-time latency analysis: traditionally, latency bounds across the memory hierarchy are obtained by summing individual bounds for each component, but GRROF allows a tighter pipelined analysis [34], [35].

In detail, the main contributions of this paper are:

- We extend Octopus to support full system simulation under the gem5 simulation platform and employing Arm architecture cores. This allows us to simulate and evaluate complex applications running under Linux, like the SLAM component presented in the Arm Industrial Challenge.
- We further integrate in our simulation framework the Arm Adaptive Traffic Profile (ATP) gem5 engine. This allows us to simulate the effect of adversarial traffic generated by accelerators and low-criticality cores on high-criticality real-time applications.
- Following our work in [34], we integrate and test GRROF in Octopus. Thanks to Octopus's modularity, we show that this can be achieved with only 150-200 lines of code.
- We evaluate the AR HUD case study from the Industrial Challenge on Octopus, using provided ATP traces to simulate aggressor workloads from GPU and DPU components. We show that by employing Octopus, we can readily recreate an architecture model that mirrors a modern, high-performance SoC, like the Jetson Xavier NX. We test using both standard high-performance arbitration policies (e.g., First-Ready First-Come-First-Serve), as well as GRROF, showing a reduction in worst-case observed latency of over 20x.

The remainder of the paper is organized as follows. In Section II, we recap the features of our proposed GRROF arbitration scheme and its architectural design. In Section III, we first provide required background on Octopus, and then discuss the extensions carried out in this paper. In Section IV, we describe the AR HUD case study from the Industrial Challenge. Then, Section V details our experimental setup using Linux on top of Octopus, and presents our evaluation results. Finally, Section VI discusses related work, and Section VII provides concluding remarks and future work.

II. GRROF: Coordinated Management of Memory Resources

Bounding memory interference in multi-core real-time systems is a long-standing challenge, especially with the complexity and aggressive optimization techniques in Commercial-Off-The-Shelf (COTS) memory hierarchies. Conventional approaches adopt an additive latency model, treating each memory component—interconnect, shared cache, memory controller—as an independent bottleneck. This model not only yields highly pessimistic latency bounds but also risks unsafety by ignoring the reordering and concurrency that characterize modern memory subsystems.

To address these limitations, in [34] we first introduced GRROF: an architecture for coordinated arbitration across the memory hierarchy that enables tight and safe end-to-end memory latency bounds. The cornerstone of GRROF is its system-wide arbitration coordination mechanism, which preserves request priorities consistently across all shared memory resources. This design makes it possible to exploit pipelining through the memory stack and apply the delay composition theorem [35]-an analysis approach initially devised for distributed pipelines of real-time jobs-to memory requests, resulting in tighter end-to-end latency bounds. In the rest of this section, we summarize the key mechanisms in GRROF, its proposed architectural implementation, and the resulting latency bounds.

A. Essence of GRROF

GRROF aims at achieving two main goals through resource management coordination:

1) ***Intra-Core Priority Preservation.*** A key challenge in analyzing memory behavior for out-of-order (OoO) cores lies in the presence of multiple inflight memory requests. Traditional arbitration mechanisms operate with only local visibility, often treating the first-arriving request as the oldest, regardless of its issuance order within the originating core. This discrepancy leads to intra-core priority inversion, where a younger request bypasses an older one due to dynamic resource contention or architectural reordering. GRROF fundamentally addresses this issue by tracking the true program order of requests from each core using per-core FIFO queues. These queues are globally visible to all arbiters, ensuring that older requests are always prioritized over younger ones, irrespective of their arrival order at any individual arbitration point.

This design guarantees that the memory request most critical to the progress of a task—the oldest active request—retains its precedence throughout its traversal of the memory hierarchy. This property is pivotal for bounding the worst-case execution time (WCET) in real-time systems, as it ensures that no artificially delayed memory access distorts the timing predictability of task execution.

2) ***Predictable and Fair Contention Resolution Among Cores.*** Beyond intra-core consistency, GRROF provides a robust mechanism to arbitrate fairly and predictably across competing cores. It adopts a Ready-First Round-Robin (RROF) arbitration scheme that blends fairness with responsiveness. Each core is assigned a position in a global round-robin order. However, advancement in this order is conditioned on readiness of the core's oldest request. If the request is not ready (e.g., awaiting resolution of earlier stages), arbitration temporarily proceeds to the next core, without permanently forfeiting the stalled core's place in the rotation.

This approach achieves three objectives simultaneously:

- It preserves fairness by ensuring that all cores receive opportunities to progress in a cyclic manner.
- It avoids blocking arbitration on stalled or dependent requests, maximizing resource utilization.
- It aligns with the delay composition theorem, which requires that the relative priorities of requests be stable across pipeline stages in order to derive tight interference bounds (see Section II-C).

The result is a globally consistent and predictably evolving arbitration regime that preserves analyzability while accommodating the concurrency of modern multicore systems.

Example: An illustrative example, adapted from [34], is presented in Figure 1. The example depicts a set of requests traversing three resources/stages: the request bus, a bank in the bankized Last-Level Cache, and a response bus. Requests are denoted as $r_{i,j}$, where i is the index of the core that issues the request, and j the order of the request among those issued by core i. We focus on request r_{11}, which targets the same bank as requests r_{21} and r_{31}. Because r_{11} suffers LLC bank interference, it arrives at the response bus after later requests r_{12} and r_{13} of the same core, which are serviced on different LLC banks with no interference. In Figure 1a, ignoring the request order within the core leads to r_{11} being served on the response bus after r_{12} and r_{13}; furthermore, due to the RR arbitration between the cores, r_{11} also suffers interference from multiple requests of cores 2 and 3. On the other hand, when applying GGROF in Figure 1b, request r_{11} is prioritized over non-oldest requests; as r_{21} and r_{31} have already completed, core 1 has the highest priority and r_{11} is executed as soon as it becomes ready on the response bus. This drastically reduces the interference suffered by the request leading to much shorter latency.

B. Proposed Architecture

We now discuss the key architectural features of GRROF; a system block diagram is provided in Figure 2. GRROF is built on the principle of distributed arbitration with centralized coordination, where each shared memory resource retains its own independent arbiter, yet all arbiters share a common, minimalistic global state to make synchronized decisions.

GRROF Engine: Global State Tracking. At the heart of the architecture lies the GRROF Engine, a centralized but lightweight control structure responsible for maintaining two critical global data structures:

- *Round-Robin Priority Queue.* This queue maintains the current scheduling order of cores. A core is inserted into this queue upon issuing a memory request and remains there until its oldest request fully retires from the system. This design ensures that the core most impacted by memory delay retains arbitration priority until its critical request is resolved.

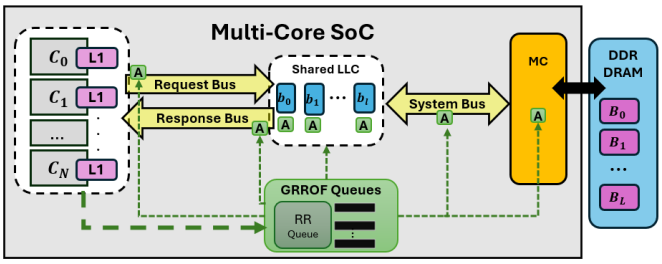
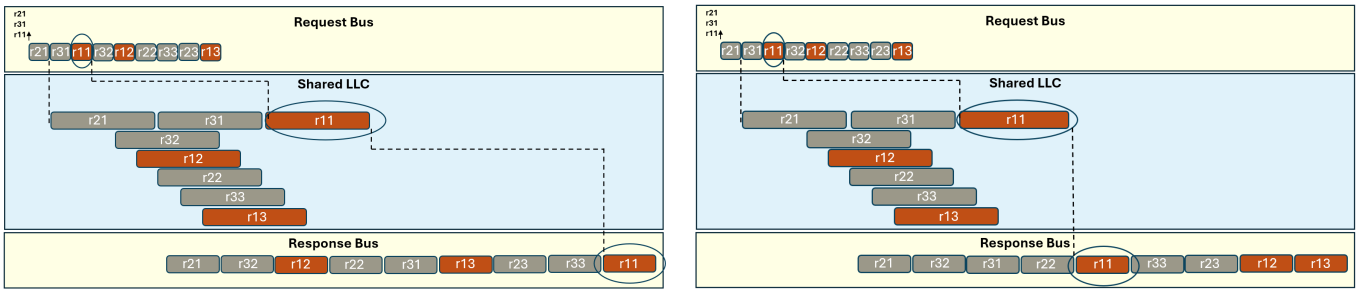


Fig. 2: The architecture of GRROF providing the global view for all arbiters.

- *Per-Core Request FIFOs.* These queues track the issuance order of each core's memory requests. Each arbiter, upon evaluating contenders for service, uses these FIFOs to determine which request is the oldest from its respective core. This allows each arbiter to apply a globally coherent ordering even with only partial visibility into the system's state.

Together, these components provide the minimal global context required for local arbiters to make decisions that respect both fairness and intra-task execution criticality.

Resource-Level Arbiter Design. Each resource in the memory hierarchy—including interconnect buses (request, response), LLC banks, system buses, and DRAM command stages—maintains a dedicated arbiter that operates in accordance with GRROF rules: The arbiter first considers ready



(a) Uncoordinated RR arbitration.

(b) Proposed GRROF solution.

Fig. 1: An example that shows the behavior of GRROF compared to separated RR. The assumed system contains three cores P_{1-3} that share resources REQ, BANK_{0-6}, and RESP. Access latencies for the REQ, BANK, and RESP resources are assumed to be 2, 25, and 5 cycles, respectively.

oldest requests, selecting among them based on the RR core order. If no oldest requests are ready, it proceeds to younger requests, again respecting per-core FIFO order and RR among cores. This two-tiered arbitration structure ensures predictable prioritization without introducing head-of-line blocking from incomplete or stalled transactions.

By maintaining this arbitration discipline at every stage, GRROF ensures that request priority remains consistent across the pipeline. This property is essential to apply pipeline-based worst-case latency analysis. Without such consistency, interference may accumulate unboundedly across stages, leading to loose or invalid latency bounds.

Coordinated DRAM Controller. GRROF's principles extend fully into the DRAM subsystem. The memory controller is designed with separate arbiters for the PRE, ACT, and CAS command stages, each governed by the global GRROF engine. Unlike conventional batching or priority inversion schemes (e.g., read-write alternation), GRROF again maintains consistent priority across DRAM stages, ensuring that memory commands issued on behalf of the oldest request are not delayed by unrelated transactions. This strict prioritization enables the memory controller to participate fully in the system-wide pipelining strategy, completing the end-to-end coordination required to tightly bound memory latency.

C. End-to-End Latency Bounds

GRROF enables the derivation of end-to-end latency bounds based on the delay composition analysis in [35], [36]. This analysis framework was initially proposed to bound the delay suffered by a distributed real-time job traversing a sequence of processors, referred to as pipeline stages, due to higher priority jobs. The key result in [35] is that the delay suffered by a job under analysis due to an interfering higher priority job is bounded by the longest execution of the higher priority job on any one stage, rather than the sum of its execution times on all stages [1]. In [39]–[41], the framework was used to derive bounds on the latency of DRAM main memory operations, by equating memory requests with real-time jobs, and DRAM commands to pipeline stages. In [34], we extended this approach to the whole memory hierarchy, again by equating hardware resources to pipeline stages. Intuitively, the delay composition theorem then ensures that each memory request is only delayed on whichever resource is slowest (i.e., the bottleneck) in the system, typically the DRAM due to inter-bank constraints between successive requests to different memory banks. As previously noted, the key result only holds if the relative priorities of memory requests is maintained across all stages, which is guaranteed by GRROF. Furthermore, due to the RR arbitration, the maximum number of interfering higher priority requests is equal to the number of interfering cores. We refer the reader to [34] for the full analysis and corresponding per-request bounds.

[1]An interesting observation is that this results is similar to the concept of "pay burst only once" in network calculus [37], [38].

III. Octopus: A Cycle-Accurate Memory System Simulator

As modern computing systems grow increasingly complex, the demand for precise and flexible simulation infrastructures has intensified—particularly for memory subsystems where cache hierarchies and interconnects play a critical role in system behavior. Accurate modeling of these subsystems is vital for performance evaluation, design space exploration, and validation of new architectural ideas. Among various simulators, many prioritize functional correctness or aggregate system behavior at a coarse granularity, limiting their suitability for detailed microarchitectural analysis. Such analysis is not only paramount for performance optimizations, but also a must for safety-critical systems controllability and observability.

To address this gap, we introduced Octopus [33] as a cache hierarchy cycle-accurate simulator. Table I summarizes Octopus features, of which we highlight four key aspects.

1) **A full cache hierarchy simulator**. Octopus modularly integrates interconnects, cache controllers, and coherence protocols: This is crucial since several of the cache hierarchy architectural optimizations involve the interaction among these components.

2) **A cycle-accurate detailed interaction modeling of modern cache hierarchies.** This includes interface buffers,

blocking and non-blocking caches, Miss-Status Handling Registers (MSHRs), write buffers, refill buffers, cache allocation policies (e.g. on miss vs on refill), and load forwarding optimizations. Previous works have shown that such detailed modeling is necessary for ensuring predictability of modern safety-critical systems, which is usually overlooked by cache analysis techniques [42].

3) **Detailed Cache Coherence modeling with all transient states.** Most performance-oriented simulators either do not model coherence at all or model it at a coarse-grained way (e.g. with only stable states or a subset of transient states). For instance, the most-detailed modeling of a coherence protocol to the best of our knowledge is in gem5's Ruby cache system, where MESI protocol is implemented using 10 total states. However, recent works on ensuring predictability in cache coherent safety-critical systems have shown that detailed transient state modeling in non-atomic and out-of-order interconnects is vital towards real-time analysis [23], [43]. For this reason, Octopus models protocols in a fine-grained way. For the MESI example, Octopus models it using 22 total states to capture all possible interactions in a complex cache hierarchy.

4) **Plug-and-Play and Extensibility by design.** One of the pain points of such detailed modeling and analysis of complex cache hierarchies is the development time of novel ideas and architectures. Octopus addresses this challenge by introducing several features that target extensibility and usability. First, cache hierarchy structure can be parameterized using only text-base configuration files as well as a single system configuration source file. Two types of configuration files are offered: XML-based, and CSV-based. A user can parameterize in these files the numbers of levels, the organization of each level, capacity, type of coherence protocol, interconnect, and arbitration. Then the system configuration source file is used as the interface glue among all the system components. Second, coherence protocols state machines are decoupled from source files. They are implemented using a tabulated csv format and read as input by the simulator. This significantly facilitates the testing, configuration, and implementation of new coherence protocols. To give quantitative measures, let us consider the implementation of the MSI protocol in both gem5 Ruby system and Octopus. To implement the MESI protocol, Ruby requires an additional 1400 lines of code (LOC), while Octopus requires only 70 lines [33].

A. Main Memory Subsystem

With covering interconnects, cache controllers, and coherence, it remains to model main memory subsystem (both memory controllers and memory devices). Towards this goal, Octopus is seamlessly integrated with MCSim [58]. Octopus interface to the main memory subsystem adopts a standard queue-based and callback interface that can be adopted to any simulator, incluing Ramulator [59] and DRAMSim [60], [61]. Nonetheless, we interface to MCSim since it aligns with the philosophy of Octopus with modularity, detailed interaction modeling, and real-time support among the design principles.

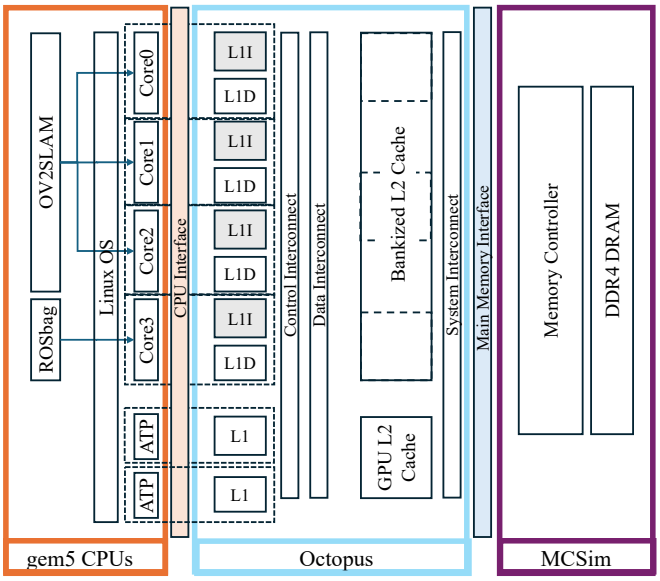


Fig. 3: System Setup

MCSim is a modular, extensible, and cycle-accurate DRAM Memory Controller (MC) simulator. The main features of MCSim are highlighted in Table II.

B. CPU and Core-Side Interfacing

As Table I highlights, Octopus supports both standalone trace-based mode as well as full-system mode. The latter is achieved through interfacing to CPU simulators. Similar to the main memory interface, Octopus provides a standard queue-based and callback interface to integrate with CPU simulators. Currently, Octopus is integrated and tested with two CPU simulators: MacSim [62] and gem5 [63].

Octopus is envisioned as an alternative to three subsystems in gem5 infrastructure: interconnects, cache system, and main memory subsystem (through MCSim). This integration is visualized in Figure 3. In this integration, Octopus+MCSim are used to model the timing behavior of memory requests from the time they are issued by the core to their private caches, to the timestamp of their return to the core.

C. Supporting Atomic Primitives in ARM ISA

To support the ARM industrial challenge, we extended Octopus to enable gem5 full system simulation using ARM cores. A key complexity is related to the way the simulation framework handles atomic instructions, which we elaborate next. The ARM Instruction Set Architecture (ISA) supports two primary mechanisms for atomic operations: **load-linked/store-conditional (LL/SC)** instructions and a more recent class of **atomic memory operations (AMOs)**. The LL/SC pair, implemented as `LDXR`/`STXR` and their variants, enables exclusive access to memory locations and is commonly used in synchronization primitives and lock-free algorithms. These instructions rely on maintaining exclusive ownership of a memory address between the load and store,

TABLE I: Octopus Simulator: Supported Configurations and Features

Category	Supported Configurations in Octopus
Coherence Protocols	• Snooping-based and directory-based protocols: MSI, MESI, MOESI • FSM-based protocol handlers support both stable and transient states • Easily extensible for new protocols, e.g., Time-Based Cache Coherence.
Interconnects	• Topologies: Point-to-Point, Unified Bus, Split Bus, Mesh, Network-on-Chip (NoC) • Configurable interconnect controllers and topology maps
Cache Hierarchy	• Arbitrary levels (L1, L2, L3...) with private/shared configurations • Multi-bank caches with non-blocking support using MSHRs and write-back buffers • Optimizations: allocate-on-fill, miss forwarding, write buffer hits
Replacement Policies	• Least Recently Used (LRU), FIFO, Random • Easily extensible to new policies
Arbiters	• FCFS, FR-FCFS, Round Robin (RR), Weighted RR, Harmonic RR, TDM • Global Coordinated Arbitration through Global Round Robin Oldest First (GRROF) [34]
Integration Modes	• Standalone mode with trace-based CPU and fixed-latency memory • Full-system simulation via integration with GEM5, MacSim, MCsim
Monitoring Tools	• Per-request detailed latency tracking • Component-specific CSV debug logs with filtering options
Configurability	• Hierarchical configuration system via CSV files and CLI • Parameters inherited, overridden, and extended modularly
Extensibility	• Modular object-oriented design with inheritance/polymorphism • Protocols and components implemented as plug-ins
Real-Time Support	• Set-partitioning • Bank Partitioning • Predictable Cache Coherent Solutions: PMSI [43], PISCOT [23],PCC [44], DUPECO [45], PENDULUM [46], PENDULUM* [47], CoHoRT [48]
Open Source	Available at https://github.com/FanosResearch/OctopusSimulator

where the success of the store is contingent on no intervening modifications.

To overcome the scalability limitations of LL/SC in large multi-core systems, ARM introduced a suite of AMO instructions as part of the Large System Extensions (LSE). This category includes instructions such as `SWP` (swap), `CAS`/`CASP` (compare-and-swap and its pair variant), and a range of atomic read-modify-write operations, including:

```
[LD|ST][ADD|EOR|CLR|SET|SMAX|SMIN|UMAX|UMIN]
```

Unlike LL/SC, AMOs are non-interruptible and execute atomically at the microarchitectural level, making them more suitable for highly parallel systems.

In our system architecture, we utilize `gem5` as the simulation framework maintaining the actual value of memory locations, with `Octopus` providing timing and coherence modeling. However, atomic instructions in `gem5` are handled by timing or atomic accesses to the main memory, while our gem5-Octopus interface uses functional access to the main memory for direct data manipulation. These functional accesses act as backdoor operations that bypass the main memory's internal logic for timing or atomic accesses, leading to incorrect modeling of atomic instructions. For example, `SWP` is treated as a naive read at gem5-Octopus interface, and exclusivity tracking for LL/SC sequences is missing.

To address this, we implement additional logic at the gem5-Octopus interface. For `SWP`, we decompose the operation into explicit read and write stages and introduce a tracking map to record the memory addresses involved. This prevents interference from interrupts and preserves atomic semantics.

For LL/SC support, we maintain a per-core exclusivity map that tracks addresses accessed via `LDXR`. The exclusivity is revoked upon any write to the address, either from the owner core or others, aligning with the ARM LL/SC semantics. This support is complicated by two key factors: (1) the "soft" nature of LL/SC, which allows some interleaving, and (2) the asynchronous behavior of Octopus, where memory requests and responses may arrive out of order.

TABLE II: Features and System Support of MCSim

Feature	Description
Operation Modes	Supports both trace-based simulation and integration with full-system simulators (gem5, MacSim, and Octopus)
Requestor Differentiation	Supports per-requestor buffers and criticality-aware scheduling
Memory Standards Support	Compatible with various DRAM standards (via integration with Ramulator); supports configurable DRAM hierarchy (channels, ranks, banks)
Extensibility	New memory controllers can be added with minimal effort; average 133 LOC per controller
Queue Configurability	Fully configurable request and command queues; supports hierarchy-level customization
Validation	Validated using MCXplore regression suite and comparison with existing simulators
Implemented High-Performance MCs	FCFS, FR-FCFS [49], FR-FCFS with Threshold [50], BLISS [51], PAR-BS [52]
Implemented Predictable MCs	REQBundle [53], CMDBundle [54], ORP [55], MEDUSA [15], RTMem [11], ROC [10], RankRe-Order [7], MCMC [5], AMC [13], DCmc [9], MAG [56], PMC [57], DRAMBulism [12]
Open Source	Available at https://github.com/uwuser/MCsim

To mitigate race conditions, we enforce the following constraints:

1) Once a `STXR` is issued to Octopus, no further `LDXR` instructions to the same address are allowed until the `STXR` response is received.
2) Regular store operations to the same address are delayed until the pending `STXR` returns from Octopus.

With extended logic, CPU memory traffic goes through the gem5-Octopus interface for atomic semantics handling first, then goes through Octopus for coherence and end-to-end timing tracking, and finally uses functional access to gem5 main memory to access the actual data.

As an alternative to functional accesses, we propose using `gem5`'s atomic memory access, which would enable reuse of the existing logic for atomic operations in the memory model. However, this approach raises potential issues due to the mixing of atomic and timing access modes, which requires further investigation to ensure correctness and consistency.

Around $\sim 1K$ lines of code are written towards interfacing Octopus to gem5 as an external component. Around 6 months of work were needed to verify full integration and Linux bring up using the ARM ISA.

D. GRROF Support

Adding the new GRROF arbiter and integrating a centralized engine into the Octopus cache simulator is both feasible and efficient, owing to Octopus's modular architecture and its seamless integration with MCsim. Octopus already models all levels of the memory hierarchy in detail—including cache hierarchies, interconnects, and coherence protocols—and supports modular arbiters that handle contention across shared resources. This design enables a unified arbitration framework that can be applied to different types of shared components (e.g., interconnects, caches, memory controllers) without rewriting arbitration logic for each. In many cases, implementing a new policy can require as little as 20 lines of C++ code, depending on its complexity, allowing new control logic to be inserted with minimal disruption. Thus, implementing the GRROF arbiter, which regulates access to shared interconnects, caches, and memory controllers, is achieved by extending existing arbitration modules. Likewise, the centralized engine is integrated by leveraging Octopus's message-passing infrastructure to track in-flight memory requests and expose their ordering information to the arbiter. In practice, this involves defining global queues (e.g., a RR buffer and per-core FIFOs) and implementing APIs to interface with arbiters, with a total code footprint of approximately 150 lines. Octopus enables low-overhead integration and validation, making it a practical and developer-friendly platform for architectural exploration and experimentation in memory subsystems.

IV. ARM INDUSTRIAL CHALLENGE: AR HUD CASE STUDY

The AR HUD case study introduced in the Arm Industrial Challenge [27] comprises two main applications: a Simultaneous Localization and Mapping (SLAM) function, and a head pose estimation function. The SLAM algorithm uses sensor inputs to simultaneously localize the vehicles and generate a map of the environment. This information is used to provide updates on vehicle status to the driver; the head pose estimation is used to adjust the projected image to the driver's viewpoint.

The SLAM implementation selected by ARM for the case study is based on $\mathrm{OV}^2\mathrm{SLAM}$ [64], a high-performance real-time visual SLAM algorithm. The application is configured to use a stereo camera and employs three real-time threads:

- Front-end Thread: The front-end thread is tasked with real-time pose estimation of the camera. This process encompasses several key tasks: image preprocessing, key-point tracking, outlier rejection, pose determination, and the initiation of keyframe creation. The thread is executed periodically, for each incoming camera frame.
- Mapping Thread: The mapping thread is in charge of processing every new keyframe to create new 3D map points by triangulation and to track the current local map in order to minimize drift. This thread is aperiodic: it only runs when a keyframe is generated.
- State Optimization Thread: The state optimization thread is in charge of running a local Bundle Adjustment to refine selected keyframes' poses and 3D map point positions. Similarly to mapping, this thread is also aperiodic.

The industrial challenge provides a reference implementation of Hopenet [65] for the head pose estimation function. Hopenet requires a monocular camera pointed towards the driver, and employs a Convolutional Neural Network (CNN)

to analyze input frames. The provided implementation uses CUDA to accelerate CNN processing. When running on an NVIDIA Jetson Nano board, the authors of [29] found that the application has low CPU utilization and mostly executes on the integrated GPU.

Finally, the industrial challenge suggests running a set of aggressor workloads, representing additional tasks that may run in the system alongside the AR-HUD application, e.g., for functionalities such as navigation, managing instruments, passenger entertainment, etc. To model such workloads in the gem5 environment, the authors provided a set of AMBA ATP traffic profiles for various aggressor tasks [27].

V. Evaluation

In this section, we describe our implementation and evaluation of the AR-HUD case study on top of Octopus+MCSim integrated with Gem5. In particular, we demonstrate how GRROF protects the oldest and most critical requests from priority inversion, while also enabling the derivation of tight yet safe end-to-end latency bounds. We first discuss the evaluation setup in Section V-A, and then present results in Section V-B.

A. Evaluation Setup

In the following, we describe in detail how we setup the virtual platform to execute the AR HUD case study from the Industrial Challenge. Note that OV^{2}SLAM relies on the Robot Operating System (ROS) [66] middleware to execute. Due to its complexity and reliance on Linux, it is not possible to run ROS in gem5 syscall emulation mode; instead, we need to perform full system simulation.

Simulated System: Figure 3 depicts a block diagram of the system. As mentioned above, the gem5 simulator is used in full system mode configured with Arm VExpress_GEM5_V1 platform and 4-cores O3CPU processor model. We model the cache hierarchy in Octopus after the Xavier platform: private 4-way 64KB L1 instruction and data caches with a shared 16-way 4MB L2 cache with 8 banks. Each L1 cache has 16 Miss Status Holding Registers (MSHRs), and can thus issue at most 16 concurrent requests. We employ MCSim to simulate a single channel DDR4 memory controller employing a DDR4_2400U_8Gb_x8 memory device.

Operating System: The disk image for full system simulation is built with Ubuntu 18.04, Linux kernel 4.14, and ROS Melodic. As part of gem5 full system simulation setup, the /sbin/init of the disk image is configured to read a user specified script upon Linux boot. From the custom script, we set up the environment and launch the OV^{2}SLAM threads. Before launching OV^{2}SLAM, we also use gem5 m5 operations to intercept the simulation process and enable Octopus logging such that we only performance monitor and report statistics for the execution of OV^{2}SLAM (region of interest) and not the Linux booting process.

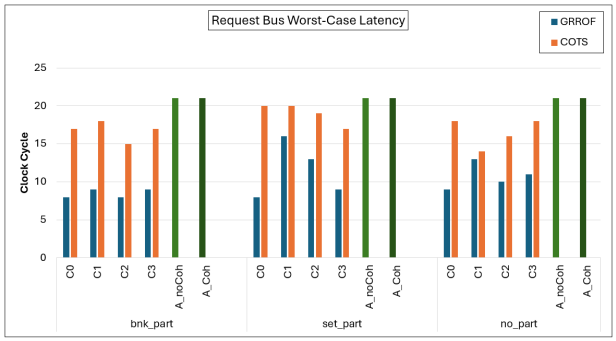
SLAM Tasks: We run OV^{2}SLAM in its fast setting, which comprises the 3 threads described in Section IV. We constrain OV^{2}SLAM to execute on cores 0-2. Input images are obtained from the EuRoC dataset [67]. For our evaluation, we employ the Machine Hall 01 scenario, which contains visual-inertial data from a drone traversing an indoor hall with various types of machinery. The data has a frequency of 20Hz. We feed images to OV^{2}SLAM using rosbag [68], a ROS tool that records or periodically playbacks data packaged in suitably prepared rosbag files. We pin rosbag to core 3.

Fig. 4: Per-Request Worst-Case Processing Latency.

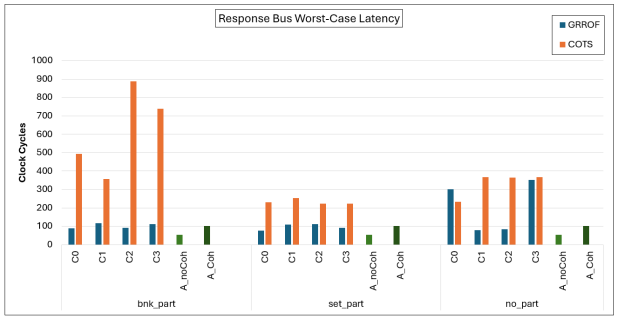
ATP: The virtual platform cannot run the reference Hopenet implementation as it does not support NVIDIA GPU simulation. Instead, we employ the Arm AMBA Adaptive Traffic Profiles (ATP) Engine [31] with provided profiles to simulate the effect of GPU and other aggressor traffic on the rest of the system. The ATP engine is a reference implementation of the AMBA ATP specification [30], which describes the dynamic behavior of a component, including the type and throughput of memory transactions. The ATP Engine provides an out-of-box gem5 adaptor exposing a CPU core-like interface for generated ATP traffic to access the memory system. We connect the ATP gem5 adaptor to Octopus in the similar manner to CPU cores with a 4-way, 64KB, 16 MSHRs data L1 for each profile. In detail, as illustrated in Figure 3, we employ two ATP profiles. The adopted ATP profiles are DPU and GPU from the GUI example in the ATP Engine repository, which randomly generate read/write operations per specified speed. To prevent the ATP Engines from overwriting physical memory addresses used by Linux and its processes, we changed the address distribution in the profiles to uniform so we can specify a specific address range.

B. Simulation Results

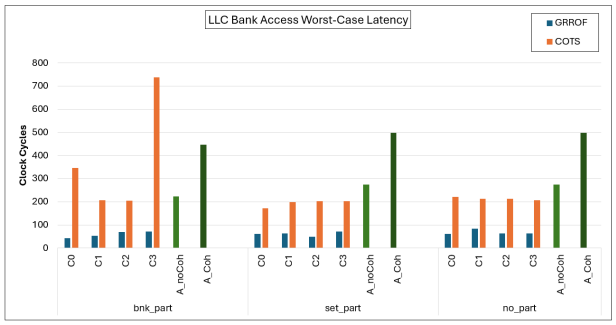
We simulated the system under two arbiter configurations: (1) using GRROF as the system-wide arbiter; (2) using standard arbiters deployed in COTS platforms, namely, First-Ready First-Come-First-Serve (FR-FCFS) arbitration at the memory controller, and standard FCFS at other resource arbiters. We also evaluated three scenarios regarding LLC isolation among cores and the ATP Engines: (A) in *bank-part*, we assign 6 LLC banks to the 4 CPU cores, and 2 LLC banks to the ATP Engines; (B) in *set-part*, we again partition the LLC in two regions, one for the cores and one for the ATP Engines, but we do so by set (i.e., cache coloring) rather than



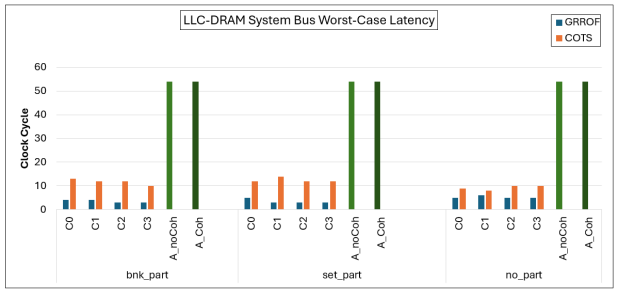
(a) Request Bus Worst-Case Latency.



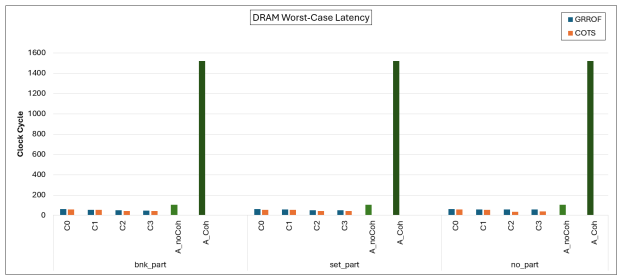
(b) Response Bus Worst-Case Latency.



(c) LLC Bank Access Worst-Case Latency.



(d) LLC-DRAM System Bus Worst-Case Latency.



(e) DRAM Access Worst-Case Latency.

Fig. 5: Worst-Case Latency for Memory Levels: both experimental and analytical values

by bank, so that both the cores and the ATP Engines are free to access all banks. (C) Finally, in *no-part*, no LLC partitioning is used.

Latency components. We use Octopus logging functionalities to log the arrival and completion time of each memory request on every memory resource. Based on the logs, we then extract the worst-case observed latency for each core at each resource. Based on the arrival time of each request in the system (the time at which the gem5 processor model presents the request to Octopus) and the time at which it retires, we also compute the processing latency and queuing latency of each requests [34], [69] (the queuing latency is the time between the request arriving in the system and becoming the oldest outstanding request of its core; the processing time is between the request becoming oldest and retiring from the system).

For analytical bounds, we show the analytically derived bounds. A_noCoh is obtained using the analysis in [34] for GRROF. The analysis in [34] derives a tight end-to-end bound for GRROF; albeit with the restriction to scenarios where cores do not share data, and as a result LLC is set-partitioned and DRAM is bank-partitioned. For this reason, we also extend these bounds to account for cache-coherent systems with shared data. These bounds are shown as A_Coh in the figures. A_Coh accounts for the fact that the OV^{2}SLAM function runs on three cores; and hence, its threads share data. This shared data triggers coherence interference such that a request from one of these cores can wait for a chain of requests to the same cache line from the other two cores. Two key modifications were made in A_Coh compared to A_noCoh: 1) Accounting for the extra interference due to coherence, and 2) applying additive delays instead of the delay composition theorem. For the first modification, the number of cores that share data is 3 and every core cannot issue several outstanding requests to the

system targeting the same cache line as they will be filtered out by the L1 MSHRs. As a result, the maximum length of the coherence dependency chain in any cache line is 3. Figure 4 depicts the worst-case observed processing latency for GRROF and FR-FCFS, as well as the analytical processing latency bounds obtained using the analysis in [34] for GRROF, for each isolation scenario and core. Note that we do not provide analytical bounds for FR-FCFS, as the policy can potentially lead to unbounded reordering; for instance, starving close-row requests in the memory controller [50].

For the second modification, while the work in [34] applied the delay composition theorem to derive a considerably tighter bound compared to additive analysis, it did so by enforcing a limitation on the system that tasks do not share data; and hence, leveraged partitioned memories. Since the SLAM function deploys three threads that run on three cores and share data, this limitation can no longer be applied. That said, A_Coh still leverages the property of GRROF of preventing intra-core priority inversion to derive tight bounds. Again, note that we cannot provide analytical bounds for DRAM with FR-FCFS arbitration, albeit bounds can be calculated for other FCFS-arbitrated resources.

In Figure 4, the observed processing latency for GRROF is lower in the partitioned cases, *bank-part* and *set-part*, compared to the *no-part* scenario, due to reduced interference from the ATP Engines. The observed GRROF latency is in all cases lower than the observed latency for FR-FCFS, up to $\sim$20x. A similar trend can be seen for most resources in Figure 5, with the exception of the DRAM resource, for which we see slightly higher latency with GRROF on some cores / scenarios. The main reason is that we found that the employed traffic profiles do not highly stress the DRAM resource; this is because the cores have a hit rate of 98% and 75% in L1 and LLC, respectively. Under such low DRAM traffic conditions, the more efficient FR-FCFS arbitration, which avoids the overhead of read-write command switching through request reordering, can achieve better latency than the pipelined GRROF solution.

1) Average-Case Performance: Real-time systems always experiences a tension between average system performance and predictability. One of the key points of GRROF is that it does not compromise average performance to improve analytical latency bounds. Figure 6 shows the average bandwidth of the different simulated systems. Two key observations to derive from Figure 6. a) While COTS solutions allows aggressors to flood the system with requests and impact the service delivered to victim cores, GRROF through its fairness mechanism regulates this behavior and hence protects the victim cores. This is the reason why GRROF solutions achieve higher bandwidth for CPUs 0–3 as well as ATP0. This comes at the expense of regulation of the aggressor ATP1. b) This type of regulation does not significantly impact overall system performance. Indeed, Figure 6 shows that GRROF achieves better overall system throughput (by a small margin $\sim 7\%$) compared to COTS solutions. This can also be attributed to GRROF ability to prohibit intra-core priority inversion and hence reduces overall CPU stall time.

VI. Related Work

A. Cache and Interconnect Simulators

Several cache system simulators have been proposed, yet many are limited in their ability to model flexible memory hierarchies, coherence protocols, or interconnect topologies. Functional simulators such as Cachegrind [70] and Dinero IV [71] emphasize functional accuracy but do not support coherence or interconnect modeling. CASPER [72] includes basic coherence modeling but lacks support for flexible interconnects. vCSIMx86 [73] focuses on x86 cache simulation but enforces a fixed hierarchy.

Cycle-accurate simulators, including Moola [74], MARSSx86 [75], McSimA+ [76], ZSim [77], and MacSim [62], provide greater timing fidelity but are often rigid and optimized for CPU-centric performance studies. Extending them for cache-centric architecture exploration or protocol modifications is often cumbersome.

While general-purpose full-system simulators like Gem5 [63] and Sniper [78] support detailed modeling of processor and memory systems, they require significant effort to experiment with different coherence mechanisms, cache replacements, or interconnect arbitration schemes. ChampSim [79] and SiNUCA [80] offer performance insights but are restricted by hard-coded configurations and lack support for extensible coherence protocol stacks.

To address these shortcomings, Octopus introduces a modular and extensible simulation framework dedicated to cache memory systems. It supports a wide range of coherence protocols (e.g., MSI, MESI, MOESI), interconnect topologies (point-to-point, bus, mesh, NoC), arbitration policies (e.g., RR, FCFS, TDM), and operates in both standalone and full-system simulation modes. Octopus is designed to interoperate with CPU and memory simulators like Gem5 and MCsim [58], enabling researchers to simulate complete memory hierarchies with high accuracy.

B. Main Memory and Memory Controller Simulators

In parallel, various DRAM device simulators have been proposed for evaluating memory subsystem performance. Popular tools such as DRAMSim2 [60], Ramulator [59], and USIMM [81] provide detailed DRAM device models and allow users to simulate specific memory standards and timing constraints. However, these tools focus exclusively on DRAM behavior and do not provide sufficient abstractions or interfaces for experimenting with memory controller (MC) designs. They also lack the ability to distinguish between different requestor classes or model criticality-aware scheduling policies.

MCsim [58] addresses this gap by offering an extensible, modular, and cycle-accurate memory controller simulation framework. It enables rapid prototyping of a wide range of memory controller policies (e.g., FR-FCFS, PARBS, ORP, BLISS) with minimal implementation effort—on average, just 133 lines of code per new controller. It supports trace-based simulation as well as integration with external simulators

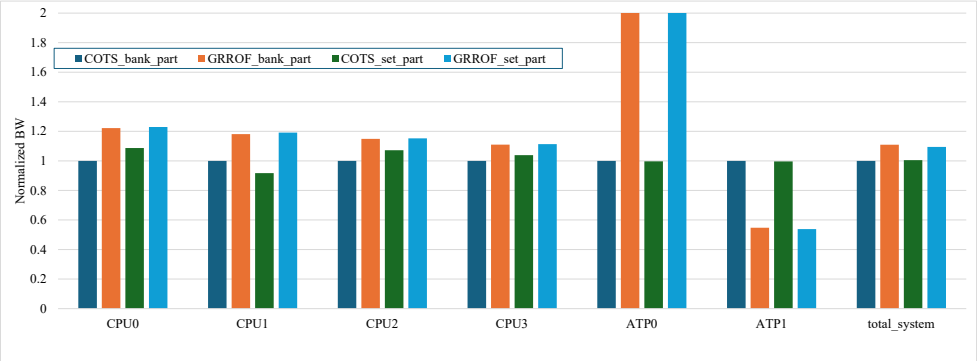


Fig. 6: Bandwidth delivered to cores and system overall bandwidth. All numbers are normalized to COTS bank_part scenario.

like gem5 and Octopus through a unified interface. MCsim supports flexible queue structures across the DRAM hierarchy (channels, ranks, banks), criticality-aware arbitration, and configurable command generation and scheduling policies.

Together, Octopus and MCsim form a comprehensive simulation stack capable of modeling complex memory hierarchies and interactions from CPU to DRAM, enabling accurate performance and worst-case latency evaluations in both real-time and high-performance computing scenarios.

C. Coordinated Global Arbitration

The additive latency model [82] is a widely used approach for estimating the total Worst-Case Execution Time (WCET) of tasks on multi-core systems. In this method, each shared resource susceptible to contention is analyzed independently to determine an upper bound on access latency. These individual bounds are then combined to derive a safe overall WCET estimate. This methodology forms the basis of most existing work on memory-related interference, with research focusing on specific components such as caches [43], [45], [83], DRAM [12], [40], [50], [84], and memory interconnects [23], [85]. To obtain end-to-end WCET, this method must be applied to each considered resource, which often results in highly pessimistic bounds when the individual delays are summed. To move beyond this decoupled analysis, [34] introduces Global Round Robin Oldest-First (GRROF), a predictable and efficient memory scheduling mechanism applied across the entire memory hierarchy. Unlike traditional approaches that treat resources in isolation, GRROF enables arbiters to function in full parallelism while remaining synchronized through a centralized coordination engine. This engine tracks and updates global state information about all in-flight requests, enabling holistic system-level coordination. Building on GRROF, this work derives a tight upper bound on the end-to-end latency experienced by a memory request as it traverses the system, from issuance to the return of data to the requesting core.

VII. Conclusions

With ever increasing complexity and heterogeneity in modern SoCs, we believe that novel architectural solutions are required to monitor the effects of resource contention across the memory hierarchy and restore timing isolation between software partitions in mixed-critical systems. Flexible tools are needed for rapid prototyping of research ideas, as well as testing and evaluation on actual realistic embedded case studies rather than isolated applications from benchmark suites. To this end, in this paper we discussed the application of Octopus, a gem5-integrated cache system simulator, to evaluate the AR HUD case study provided by the Arm Industrial Challenge [27]. We extended Octopus to support full system simulation with out-of-order Arm cores, as integrated the ATP Engine to simulate aggressor traffic based on profiles provided in the Industrial Challenge. We faithfully replicated the cache architecture of a modern embedded SoC, and tested the effects of resource contention using both standard, high-performance resource arbiters, as well as our recently proposed GRROF arbitration framework [34] that provides end-to-end latency guarantees to memory requests. Simulation results show that GRROF can greatly reduce the observed worst-case request latency, while maintaining comparable performance in the average case.

Octopus is open source and publicly available at: https://github.com/FanosResearch/OctopusSimulator. We will continue supporting both Octopus and our employed memory controller simulator, MCSim, to to support both real-time systems research and broader architectural exploration. We also seek to further extend Octopus towards full simulation of heterogeneous systems, including accelerators. In particular, we plan to integrate the gem5 AMD GPU model [86], which would allow us to run the Hopenet Head Pose Estimation application [87] on the simulated GCN3 GPU using the Radeon Open Compute platform (ROCm), rather than injecting GPU-like traffic profiles through the ATP.

Acknowledgments

We would like to thank the authors of [29] for open-sourcing the code used in their paper to run the OV^{2}SLAM case study on ROS; this helped us to build our simulation disk image.

This work has been supported in part by NSERC. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "An empirical survey-based study into industry practice in real-time systems," in *2020 IEEE Real-Time Systems Symposium (RTSS)*. York, 2020.

[2] P. D. Groves, *Principles of GNSS, inertial, and multi-sensor integrated navigation systems*, ser. GNSS technology and applications series. Artech House, 2008.

[3] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "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. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

[4] B. Akesson and K. Goossens, *Memory controllers for real-time embedded systems*. Springer, 2011.

[5] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst, "A mixed critical memory controller using bank privatization and fixed priority scheduling," in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2014, pp. 1–10.

[6] L. Ecco and R. Ernst, "Improved DRAM Timing Bounds for Real-Time DRAM Controllers Read/Write Bundling," in *Real-Time Systems Symposium*, 2015.

[7] L. Ecco, A. Kostrzewa, and R. Ernst, "Minimizing DRAM Rank Switching Overhead for Improved Timing Bounds and Performance," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[8] D. Guo and R. Pellizzoni, "A requests bundling dram controller for mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*. IEEE, 2017, pp. 247–258.

[9] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and F. J. Cazorla, "A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study," in *2014 IEEE Real-Time Systems Symposium*, 2014.

[10] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni, "A rank-switching, open-row dram controller for time-predictable systems," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 27–38.

[11] Y. Li, B. Akesson, and K. Goossens, "Dynamic command scheduling for real-time memory controllers," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 3–14.

[12] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "Drambulism: Balancing performance and predictability through dynamic pipelining," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[13] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time cmps," *IEEE Embedded Systems Letters*, vol. 1, no. 4, pp. 86–90, 2009.

[14] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: bank privatization for predictability and temporal isolation," in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '11. New York, NY, USA: ACM, 2011, pp. 99–108.

[15] P. K. Valsan and H. Yun, "MEDUSA: a predictable and high-performance DRAM controller for multicore based embedded systems," in *Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2015, pp. 86–93.

[16] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, pp. 1–36, 2015.

[17] M. Hassan, A. M. Kaushik, and H. Patel, "Predictable cache coherence for multi-core real-time systems," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2017, pp. 235–246.

[18] N. Sritharan, A. Kaushik, M. Hassan, and H. Patel, "Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 433–445.

[19] A. M. Kaushik, P. Tegegn, Z. Wu, and H. Patel, "Carp: A data communication mechanism for multi-core mixed-criticality systems," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 419–432.

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

[21] A. M. Kaushik and H. Patel, "A systematic approach to achieving tight worst-case latency and high-performance under predictable cache coherence," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 105–117.

[22] M. Hassan, "Discriminative coherence: Balancing performance and latency bounds in data-sharing multi-core real-time systems," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[23] S. Hessien and M. Hassan, "The best of all worlds: Improving predictability at the performance of conventional coherence with no protocol modifications," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020, pp. 218–230.

[24] "Memory system resource partitioning and monitoring (mpam), for a-profile architecture," https://developer.arm.com/documentation/ddi0598/latest/, Arm Ltd., 2024, accessed: 2024-11-30.

[25] "Intel® resource director technology (intel® rdt) framework," https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html, Intel Corporation, 2024, accessed: 2024-11-30.

[26] Advanced Micro Devices, Inc., "Amd64 technology platform quality of service extensions," Advanced Micro Devices, Inc., Tech. Rep. 56375 Rev. 1.03, 2021, accessed: 2025-06-02. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/other/56375_1_03_PUB.pdf

[27] M. Andreozzi, G. Gabrielli, B. Venu, and G. Travaglini, "Industrial challenge 2022: A high-performance real-time case study on arm," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, 2022.

[28] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[29] M. Bechtel and H. Yun, "Analysis and mitigation of shared resource contention on heterogeneous multicore: An industrial case study," *IEEE Transactions on Computers*, 2024.

[30] Arm Ltd., "Amba adaptive traffic profiles specification," https://developer.arm.com/documentation/ihi0082/latest/, 2019, aRM IHI 0082A, Accessed: 2025-06-01.

[31] M. Andreozzi and A. Herrera, "Amba adaptive traffic profiles (atp) engine," https://github.com/ARM-software/ATP-Engine, 2020, accessed: 2025-06-01.

[32] "Gem5 ruby memory system." [Online]. Available: https://www.gem5.org/documentation/general_docs/ruby/

[33] M. Hossam, S. Hessien, and M. Hassan, "Octopus: a cycle-accurate cache system simulator," *IEEE Computer Architecture Letters*, 2024.

[34] S. Abdelhalim, D. Germchi, M. Hossam, R. Pellizzoni, and M. Hassan, "A tight holistic memory latency bound through1 coordinated management of memory resources," in *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, 2023.

[35] P. Jayachandran and T. Abdelzaher, "Delay composition in preemptive and non-preemptive real-time pipelines," *Real-Time Systems*, vol. 40, no. 3, pp. 290–320, 2008.

[36] ——, "End-to-end delay analysis of distributed systems with cycles in the task graph," in *2009 21st Euromicro Conference on Real-Time Systems*. IEEE, 2009, pp. 13–22.

[37] J.-Y. L. Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, ser. Lecture Notes in Computer Science. Springer, 2001, vol. 2050, accessed: 2025-06-02. [Online]. Available: https://link.springer.com/book/10.1007/3-540-45318-0

[38] A. Bouillard, M. Boyer, and E. L. Corronc, *Deterministic Network Calculus: From Theory to Practical Implementation*. Wiley, 2018.

[39] H. Yun, R. Pellizzoni, and P. K. Valsan, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2015, pp. 184–195.

[40] M. Hassan and R. Pellizzoni, "Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[41] ——, "Analysis of memory contention in heterogeneous cots mpsocs," in *32nd Euromicro Conference on Real-Time Systems (ECRTS)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 1–24, accessed: 2025-06-02. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12383/

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

[43] A. M. Kaushik, M. Hassan, and H. Patel, "Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems," *IEEE Transactons on Computers (TC)*, pp. 1–23, 2020.

[44] M. Hossam and M. Hassan, "Predictably and efficiently integrating cots cache coherence in real-time systems," in *34th ECRTS*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022.

[45] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "Parallelism-Aware High-Performance Cache Coherence with Tight Latency Bounds," in *34th Euromicro Conference on Real-Time Systems (ECRTS)*, 2022, pp. 16:1–16:27.

[46] N. Sritharan *et al.*, "Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems," in *IEEE RTSS*, 2019.

[47] S. Bayes, M. Hossam, and M. Hassan, "Shared data kills real-time cache analysis. how to resurrect it?" in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[48] S. Bayes and M. Hassan, "Criticality and requirement aware heterogeneous coherence for mixed criticality systems," in *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2025, pp. 1–7.

[49] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *ACMSIGARCH Computer Architecture News*, 2000.

[50] M. Hassan and R. Pellizzoni, "Analysis of memory-contention in heterogeneous cots mpsocs," in *Euromicro Conference on Real-Time Systems*, 2020.

[51] L. Subramanian *et al.*, "Bliss: Balancing performance, fairness and complexity in memory access schedyuling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, pp. 3071–3087, 2016.

[52] O. Mutlu *et al.*, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 2008.

[53] D. Guo and R. Pellizzoni, "A requests bundling dram controller for mixed-criticality systems," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2017, pp. 247–258.

[54] L. Ecco and R. Ernst, "Improved dram timing bounds for real-time dram controllers with read/write bundling," in *2015 IEEE Real-Time Systems Symposium*. IEEE, 2015, pp. 53–64.

[55] Z. Wu, Y. Krish, and R. Pellizzoni, "Worst-case analysis of dram latency in multi requestor systems," in *IEEE 34th Real-Time Systems Symposium*, 2013.

[56] H. Kim *et al.*, "A predictable and command-level priority-based DRAM controller for mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015, pp. 317–326.

[57] M. Hassan, H. Patel, and R. Pellizzoni, "A framework for scheduling dram memory accesses for multi-core mixed-time critical systems," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015, pp. 307–316.

[58] R. Mirosanlou *et al.*, "Mcsim: An extensible dram memory controller simulator," *IEEE Computer Architecture Letters*, 2020.

[59] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2015.

[60] P. Rosenfeld *et al.*, "Dramsim2: A cycle accurate memory system simulator," *IEEE computer architecture letters*, vol. 10, no. 1, pp. 16–19, 2011.

[61] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: A cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.

[62] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework user guide," *Georgia Institute of Technology*, 2012.

[63] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, 2011.

[64] M. Ferrera, A. Eudes, J. Moras, M. Sanfourche, and G. L. Besnerais, "Ov^{2}slam : A fully online and versatile visual slam for real-time applications," 2021.

[65] N. Ruiz, E. Chong, and J. M. Rehg, "Fine-grained head pose estimation without keypoints," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.

[66] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009. [Online]. Available: http://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf

[67] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. W. Achtelik, and R. Siegwart, "The euroc micro aerial vehicle datasets," *The International Journal of Robotics Research*, vol. 35, no. 10, pp. 1157–1163, 2016. [Online]. Available: https://www.researchgate.net/publication/291954561_The_EuRoC_micro_aerial_vehicle_datasets

[68] T. Field, J. Leibs, J. Bowman, and D. Thomas, "rosbag - recording and playback of ros topics," https://wiki.ros.org/rosbag, accessed: 2025-06-01.

[69] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "Parallelism-aware high-performance cache coherence with tight latency bounds," in *Euromicro Conference on Real-Time Systems (ECRTS 2022)*, 2022.

[70] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, 2007.

[71] J. Edler, "Dinero iv: Trace-driven uniprocessor cache simulator," *http://www. cs. wisc. edu/~ markhill/DineroIV*, 1994.

[72] R. Iyer, "On modeling and analyzing cache hierarchies using casper," in *11th IEEE/ACM MASCOTS 2003*. IEEE, 2003.

[73] H. Kang and J. L. Wong, "vcsimx86: a cache simulation framework for x86 virtualization hosts," *Stony Brook University*, 2013.

[74] C. F. Shelor and K. M. Kavi, "Moola: Multicore cache simulator," in *30th CATA*, 2015.

[75] A. Patel, F. Afram, and K. Ghose, "Marss-x86: A qemu-based microarchitectural and systems simulator for x86 multicore processors," in *1st International Qemu Users? Forum*. Citeseer, 2011.

[76] J. H. Ahn *et al.*, "Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *2013 IEEE ISPASS*. IEEE, 2013.

[77] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer architecture news*, 2013.

[78] T. E. Carlson *et al.*, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[79] N. Gober *et al.*, "The championship simulator: Architectural simulation for education and competition," *arXiv preprint arXiv:2210.14324*, 2022.

[80] M. A. Z. Alves *et al.*, "Sinuca: A validated micro-architecture simulator," in *2015 IEEE 17th International Conference on High Performance Computing and Communications*. IEEE, 2015.

[81] N. Chatterjee *et al.*, "Usimm: the utah simulated memory module," *University of Utah, Tech. Rep*, 2012.

[82] S. Hahn, M. Jacobs, and J. Reineke, "Enabling compositionality for multicore timing analysis," in *International conference on real-time networks and systems (RTNS)*, 2016.

[83] M. Hossam and M. Hassan, "Predictably and efficiently integrating cots cache coherence in real-time systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2022, pp. 17:1–17:23.

[84] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "Duetto: Latency guarantees at minimal performance cost," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1136–1141.

[85] M. Hassan and H. Patel, "Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–11.

[86] "Gem5 gcn3 gpu model." [Online]. Available: https://www.gem5.org/documentation/general_docs/gpu_models/GCN3

[87] N. Ruiz, E. Chong, and J. M. Rehg, "Fine-grained head pose estimation without keypoints," 2018.