A Tight Holistic Memory Latency Bound Through Coordinated Management of Memory Resources

Shorouk Abdelhalim ✉
McMaster University, Hamilton, Canada

Danesh Germchi ✉
University of Waterloo, Canada

Mohamed Hossam ✉
McMaster University, Hamilton, Canada

Rodolfo Pellizzoni ✉
University of Waterloo, Canada

Mohamed Hassan ✉
McMaster University, Hamilton, Canada

Abstract

To facilitate the safe adoption of multi-core platforms in real-time systems, a plethora of recent research efforts aim at bounding the delays induced by interference upon accessing the shared memory resources in these platforms. These efforts, despite their value, are scattered, with each one focusing solely on only one of these resources with the premise that latency bounds separately driven for each resource can be added all together to provide a safe end-to-end memory bound. In this work, we put this assumption to the test for the first time by 1) considering a realistic multi-core memory hierarchy system, 2) deriving the bounds for accessing the shared resources in this system, and 3) highlighting the limitations of this widely-adopted approach. In particular, we show that this approach leads to not only excessively pessimistic but also unsafe bounds. Motivated by these findings, we propose GRROF: a novel approach to predictably and efficiently schedule memory requests while traversing the entire memory hierarchy through coordination among arbiters managing all the resources in this hierarchy. By virtue of this novel mechanism, we managed to exploit pipelining upon analyzing the latency of the memory requests for tightly bounding the worst-case latency. We prove in the paper that GRROF enables us to derive a drastically tighter bound compared to the common additive latency approach with more than 18× reduction in the end-to-end memory latency bound for a modern Out-of-Order quad-core platform. The reduction is further improved significantly with the increase in the number of cores. The proposed solution is fully prototyped and tested in a cycle-accurate simulation. We also compare it with real-time competitive state-of-the-art and performance-oriented solutions existing in modern Commercial-off-the-Shelf (COTS) platforms.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture; Computer systems organization → Embedded hardware

Keywords and phrases Predictability, Main Memory, Caches, Real-time

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.17

Supplementary Material *Software (Source Code)*: `https://gitlab.com/FanosLab/endtoend_wcl_cases_matlab/`

Funding This work has been supported in part by NSERC, CMC Microsystems, and TII. 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.

Acknowledgements We would like to thank the anonymous reviewers for their valuable feedback, and our shepherd for helping to significantly improve this paper.

© Shorouk Abdelhalim, Danesh Germchi, Mohamed Hossam, Rodolfo Pellizzoni, and Mohamed Hassan;
licensed under Creative Commons License CC-BY 4.0

1 Introduction

With the increasing performance requirements and amounts of data to be processed by modern real-time systems, adopting multi-core platforms becomes favorable, if not necessary. One of the main roadblocks to this adoption is the architectural complexity of these platforms, which threatens the timing analyzability of the system and the ability to derive safe yet tight Worst-Case Execution Time (WCET) for real-time tasks. In particular, the several memory resources among the cores pose a significant challenge in bounding the interference-induced delays suffered upon accessing these resources. Therefore, the real-time community has recently invested significant (but somewhat scattered) efforts to address this challenge at each of the different memory resources, including interconnects [14,25,42], caches [8,11,27,28,37,41], and main memories [1,6,9,20]. Each of these efforts focused solely on one of these resources with the premise that latency bounds separately driven for each resource can be added altogether to provide a safe end-to-end memory bound, which we refer to as the *additive latency* approach [10].

In this work, we assess this assumption by conducting the following contributions.

1. We consider a comprehensive and realistic multi-core memory hierarchy system modeled after Commercial-off-the-Shelf (COTS) platforms, where they are independent resources that can be accessed in parallel. This includes a split-transaction interconnect between private L1 caches and the shared Last Level Cache (LLC) composed of a request and a response bus, a realistic cache model with write buffers and non-blocking support to enable several requests to be serviced in parallel, a bankized LLC with several independent banks that can be accessed in parallel, and a system bus to carry requests from the LLC misses and write backs to the memory controller to be sent to the off-chip memory. Section 3 elaborates on this system model.
2. We derive the bounds for the resources in this system following the aforementioned additive latency approach by considering each resource independently. We then use this step to highlight two limitations of this approach. In particular, a) on the one hand, it leads to excessively pessimistic bounds to the level that they reach several thousands of cycles for one request and hence becomes practically useless. This is due to the aggressive reordering and parallelism deployed in these COTS platforms. b) On the other hand, it leads to unsafe bounds due to the fact that each separate resource has to make local assumptions about the ordering of requests that does not necessarily align with the actual request orders in the system. These two limitations are further discussed in Section 2.2 and are illustrated with a numerical example in Section 4.
3. To address this problem, we propose the Global Round Robin Oldest-First (GRROF) as a novel methodology to predictably and efficiently schedule memory requests upon traveling through the entire memory hierarchy. Instead of managing resources independently and analyzing them in isolation from each other, GRROF enables arbiters to operate fully in parallel yet coordinate by sharing and updating a centralized engine that tracks states about requests currently in the system. More details about this approach are provided in Section 4.
4. We use this coordination to conduct a novel analysis that derives a bound, for the first time to the best of our knowledge, on the end-to-end latency suffered by a request upon accessing the memory hierarchy including accessing the interconnect, the LLC, the system bus to the off-chip DRAM, if it is a miss in the LLC, and all the way until it returns to its requesting core and retires. This novel analysis leverages GRROF to pipeline requests among these resource and apply the delay composition theorem originally proposed in [21].

Table 1 DDR4-2400U Timing Constraints [38]. l and s refer to the large (same bank group) and small (different bank groups) timing constraints, respectively.

	Inter-Bank Constraints			Intra-Bank Constraints	
	Description	Cycles		Description	Cycles
t_{RRD}	ACT to ACT	l=6, s=4	t_{RL}	RD to DATA	18
t_{FAW}	4 ACT Window	26	t_{WL}	WR to DATA	12
t_{WTR}	WR DATA to RD	l=9, s=3	t_{WR}	WR DATA to PRE	18
t_{WtoR}	WR to RD	25	t_{RP}	PRE to ACT	18
t_{RTW}	RD to WR	12	t_{RCD}	ACT to CAS	18
t_{BUS}	DATA	4	t_{RTP}	RD to PRE	9
t_{CCD}	CAS to CAS	l=6, s=4	t_{RC}	ACT to ACT	57
			t_{RAS}	ACT to PRE	39

This leads to a drastically tighter bound compared to the common additive latency approach. This reaches more than 18× reduction in a modern Out-of-Order quad-core platform. The analysis is derived in Section 5.

5. We prototype this whole memory system in a cycle-accurate simulator in addition to three other approaches. The first two are modeled after predictable hard-ware real-time solutions, such as Round-Robin (RR) [7,36] and Round-Robin Oldest-First (RROF) [30,33], while the third represents a First-Ready First-Come First-Serve (FRFCFS) approach that reorders requests to increase system performance and is commonly used in COTS platforms. Section 6 discusses the detailed results of these comparisons. And finally, Section 7 is the conclusion.

2 Background and Related Work

2.1 DRAM Memory Background

DRAM device is the off-chip main memory that communicates with on-chip processing elements through a Memory Controller (MC). The device consists of multiple banks of 2D array structure that are indexed by row and column addresses and accessed through data, address, and command buses. Accessing data from a DRAM bank is generally a two-stage process. 1) The row address is provided to activate the requested row through an activation (ACT) command. 2) The column address is provided to conduct the requested read/write operation through a CAS (RD/WR) command. Each DRAM bank also has a row buffer that holds the most recently accessed row from that bank. This enables future accesses to the same row by read/write from the buffer directly without re-activating the row (row hit), and that only requires a CAS command. On the other hand, if a request accesses a row different than the one in the buffer (row miss), the MC has first to pre-charge the row through a PRE command, and then issues the ACT and CAS commands. Those commands (PRE, ACT, and R/W CAS) should be separated by the timing constraints defined in the DRAM JEDEC standard [38] to ensure a correct behavior from the DRAM. Table 1 shows the relevant timing constraints. Some of these constraints apply to the commands of the same bank (intra-bank), while others apply to the commands among different banks (inter-bank). A command is considered intra-ready or inter-ready when it satisfies its intra-bank or inter-bank constraints, respectively, and it becomes ready when both constraints are met.

2.2 Motivation: State-of-The-Art Limitations

The current paradigm to calculate the total WCET of a task in a multi-core platform while accounting for the interference along the memory hierarchy is to use the additive latency approach [10]. In this approach, every resource that is subject to contention is analyzed separately (i.e., independent of other resources) to derive an upper bound on the latency suffered upon accessing that resource. Afterwards, all latency bounds of all resources can be added together to provide an overall safe bound. Most of the existing work in bounding memory-related interference follows that approach; for instance, by focusing on caches [12, 13, 18, 23, 33], DRAM [15, 16, 30, 31], or memory interconnect [14, 17]. Thus, we make two critical observations about this approach.

1) On the one hand, **this analysis conducted separately at each resource has to assume the maximum possible interference at this resource**. This has to be applied to all considered resources leading to very pessimistic bounds when all added together. For example, for a multi-core system with M Out-of-Order (OoO) cores, each of which can have N_{pend} possible outstanding requests, the analysis has to assume the maximum possible interfering requests from all other cores on the resource under analysis, which is $(M-1) \cdot N_{pend}$. For example, existing work in analyzing DRAMs has considered this number of possible competing requests [15, 43]. We make the observation that this is due to the fact that *COTS platforms, to optimize performance, deploy aggressive parallelism among these resource and reorderings among requests targeting them. They do not maintain a global ordering view that is shared by all these resources.* Using this observation, we show that by providing such global ordering, GRROFenables us to derive a considerably tighter bound by making a holistic analysis of all the resources amenable. 2) On the other hand, **the conducted analysis considering only one resource can lead to unsafe assumptions**. In particular, in the case of analyzing requests from an OoO core with multiple outstanding requests, the analysis has to consider the request that arrives first to the resource under consideration to be the oldest from that core. This is, for example, what is conducted in the existing analysis for DRAMs [32] and caches [33]. Although this is true from this resource perspective, it is not necessarily valid from the real (core issuance) perspective. For instance, in a real multi-core platform, where there exists parallelism in the memory hierarchy, a younger request can arrive at a resource before an older one from the same core. This simply destroys the notion of older request from a core perspective, which can entail significant delays to that request upon being arbitrated at one of the resources. The only way to derive a safe bound on such a case is to always assume that the request under analysis arrives at this resource last after the maximum possible number of earlier requests from the same core. This further pushes the pessimism of the analysis leading to extremely significant latency bounds. We will discuss these limitations more with an illustrative example in Section 4 and analytically bound the delays using the additive latency approach in Section 5.3.

2.3 Delay Composition Theorem for Real-Time Pipelines

Our analytical bounds use the delay composition strategy first introduced in [21] and apply it to the whole memory hierarchy. While this analysis method has been introduced to compute the worst-case latency of distributed real-time jobs, it has also been previously applied to obtain upper latency bounds for DRAM requests [15, 16, 43]. In detail, the analysis considers a set of jobs, which we will equate to hardware requests, executing on a given sequence of resources (or pipeline stages). Each request has a known worst-case execution time on each stage; once a request finishes executing on a stage, except the last, it immediately

becomes ready for the next stage. The total latency of a request is defined by the difference between the time it finishes execution on the last stage in the pipeline, and the time at which it arrives on the first stage. At each stage, requests are scheduled according to either a fixed-priority preemptive or a fixed-priority non-preemptive policy; in this paper, we employ the latter since it matches the behavior of hardware resources. The main result in [21] is that pipelining allows us to constrain the interference caused by higher priority requests on a given request *u*nder *a*nalysis r_{ua}: such interference is limited to the longest execution time of a higher priority request r on any stage, rather than the sum of r's execution times on all stages. Intuitively, the idea is that once r "gets ahead" of r_{ua}, it will become ready and start executing on successive stages ahead of it; therefore, it cannot cause maximum interference on r_{ua} on each stage.

However, for this property to hold, the theorem requires the relative priority of requests to remain the same on all stages: if r temporarily drops its priority below that of r_{ua} on a stage, it might be delayed by other lower-priority requests on that stage; and once it regains its higher priority on a later stage, it might interfere again with r_{ua}. One of the main contributions of this paper is proposing a novel architecture that enables the coordinated management of all memory resources (i.e., stages) such that this property is satisfied. As a result, this enables us to leverage the pipelining feature from the theorem to derive a significantly tighter holistic memory latency bound.

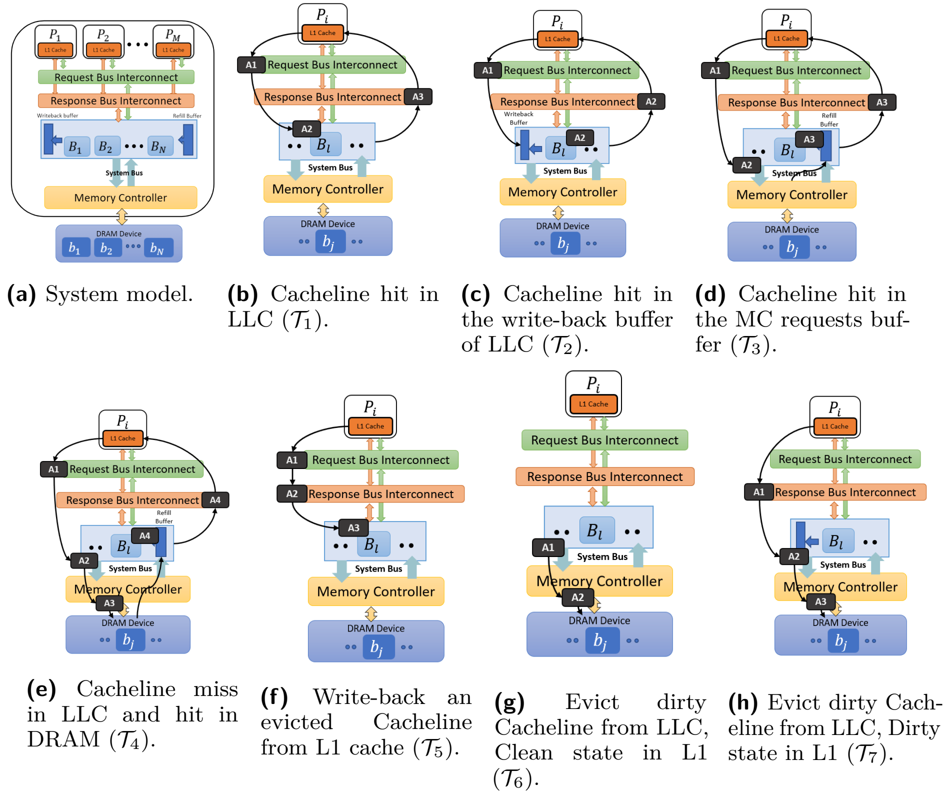
Finally, the original delay composition analysis in [21] assumes that all requests traverse the same pipeline stages. This is not generally the case for a modern memory hierarchy where requests of different types can access resources in a different order. Consider, for example, a demand miss request from a core compared to a write-back request of an evicted line from either L1 or LLC. We will thus use the improved analysis in [22], which supports such an extension. The key idea in the analysis is to split every higher priority interfering request into a set of *segments*: each segment represents the execution of that request on a sequence of consecutive stages in the path of the request under analysis, encountered either in the same or exactly in reverse order.

3 System Model

3.1 Architecture

This section introduces the hardware architecture considered in this paper, as shown in Figure 1a.

- **Processing Cores.** We consider a multi-core system with M cores $P_1, .., P_i, .., P_M$, which can be In-order (IO) or OoO cores. OoO cores can have multiple outstanding memory requests. We denote the maximum number of such in-flight requests as N_{pend}, which is usually determined by the number of available entries in the Miss Status Holding Registers (MSHRs) in the platform's caches [41, 43].
- **Caches.** We assume each core has exclusive access to a private cache (L1), and all the cores share an on-chip Last-Level Cache (LLC). In line with related work, we assume a partitioned LLC to eliminate data interference between cores at the LLC level [8]. Our proposal does not require a particular technique for partitioning; however, for the analysis, we assume a set-partitioned LLC. L1s and LLC are write-back write-allocate caches and implement Least Recently Used (LRU) replacement policy. Unlike state-of-the-art works in cache analysis in the real-time domain, a critical aspect of this paper is that we consider a more realistic cache model that employs several of the optimizations commonly deployed in COTS cache systems to improve system performance. 1) ***MSHRs.*** In order to leverage



(a) System model.

(b) Cacheline hit in LLC ($\mathcal{T}_1$).

(c) Cacheline hit in the write-back buffer of LLC ($\mathcal{T}_2$).

(d) Cacheline hit in the MC requests buffer ($\mathcal{T}_3$).

(e) Cacheline miss in LLC and hit in DRAM ($\mathcal{T}_4$).

(f) Write-back an evicted Cacheline from L1 cache ($\mathcal{T}_5$).

(g) Evict dirty Cacheline from LLC, Clean state in L1 ($\mathcal{T}_6$).

(h) Evict dirty Cacheline from LLC, Dirty state in L1 ($\mathcal{T}_7$).

Figure 1 Considered system model (a) and different request types for both demand (b – e) and write-back (f – h) requests.

the performance gain from the OoO cores, we also consider a non-blocking cache that can service multiple requests at a time. Non-blocking cache is an old concept [26] that is widely adopted in modern COTS platforms. The considered caches allow both a hit-over-miss and a miss-over-miss (i.e., servicing hits and misses while there is a pending miss), subject to the number of MSHR entries. 2) ***Allocate-on-Fill.*** We consider an allocate-on-fill cache, where cache lines that miss in the cache are allocated in the data array only upon receiving the data from the lower memory. This avoids unnecessary early eviction of cache lines and hence can enable more hits [2]. 3) ***Write Buffer.*** Caches employ a write buffer where it places dirty lines upon eviction to be written to the lower memory level. This enables a faster allocation for the new cache line without waiting for the evicted line to be written to that lower memory. Similar to MSHR, write buffering is a standard technique in modern COTS platforms [39] including Intel's [19] and ARM's [3]. It does not only exist in high-end platforms, but even the low-end ones usually used in real-time embedded systems such as the ARM M4 includes some form of a write buffer [4]. 4) **Write Buffer Hits.** Furthermore, the cache controller allows hitting in its write-back buffer if it receives a request to a dirty cache line that exists in the write buffer; hence, preventing unnecessary high miss latency. Upon hitting in the write buffer, the cache controller takes two simultaneous actions to serve such requests: it sends the requested data to the requestor and saves the data again in the cache's data array. 5) ***Bankized LLC.*** We consider a multi-bank LLC where data is distributed over N cache banks; $B_1, .., B_N$. LLC banks are independent and can serve different requests in parallel; hence,

they are modeled as separate resources, and the process time of the data in/out the data arrays of a bank is c_{BANK}. Since each LLC bank is independent, it will have its own set of write buffer and MSHRs. We assume that both write buffers and MSHRs in shared LLC are not source of interference among cores. This is because their sizes in COTS platforms are usually set large enough to accommodate the maximum number of possible outstanding requests from all cores in the system. Even in the case that this assumption does not hold for some particular platform, existing works can be used to eliminate the effect on request latency such as the work in [41] for MSHRs and in [5] for write buffers. 6) ***Miss Forwarding.*** The caches deploy a common optimization to reduce the miss penalty, where requests can be determined to be whether a hit or a miss by checking the tag/status bits. This tag checking can be done in parallel to and independent of accessing the cache data array. Consequently, upon a miss, the cache controller forwards the miss to be filled from the lower level memory without accessing the data array, which reduces miss penalty. Additionally, once the data refill arrives from the lower memory, it can be immediately forwarded to the requesting core on the response bus (subject to arbitration as detailed later on), while simultaneously placed to be written also to the cache data array. 7) ***Immediate Back Invalidation.*** We also assume that back invalidation from a lower level of memory uses a dedicated special bus, and hence, do not interfere with demand requests on the request bus.

- **Interconnect Bus.** The system model considers a split-transaction shared bus between the L1 caches and the LLC. This bus comprises two independent buses: a ***request bus*** for sending requests from the L1s to the LLC and a ***response bus*** for data transmissions. This architecture allows a concurrent operation for the requests and data responses on the buses, where the transmission latency of packets on the request and the response buses are c_{REQ} and c_{RESP}, respectively.
- **System Bus.** The system bus is the interconnect between the LLC and the main memory, and used for transmitting requests and data. A packet on the bus can contain a request, data, or both, and its transmission latency is c_{SBUS}. In our model, we assume a full-duplex system bus, as shown in Figure 2, which consists of two buses: one is for packets that are sent from LLC to the main memory (LLC-DRAM bus), and the second is for the way back from the main memory (DRAM-LLC bus).
- **Memory Controller and DRAM.** Accesses to the off-chip DRAM memory are managed through an on-chip MC, as explained in Section 2.1. The MC stores incoming requests from the system bus in per-requestor buffers. We consider a DRAM with n private banks: $b_1, .., b_n$, where the MC maps every request to a bank that is assigned to its core. Afterwards, the MC translates each request into its corresponding set of commands and buffers them into the per-bank command queues. The MC arbitrates between the ready commands based on the arbitration scheme order. Two more COTS features we consider in our system model. 1) **Write Data Queue Hit.** We assume that the MC allows demanding requests to hit in its request data buffers in order to reduce the memory latency. This means that if a demanding request reaches the request buffers of the MC while the required data is in one of the outstanding write requests, the data is read from the buffers directly. 2) **Clock Domain Crossing (CDC) Effect.** Since off-chip DRAM can generally operate at a different frequency than the on-chip core one, in our end-to-end latency calculations, we have to consider the clock domain crossings that requests suffer upon traversing the memory hierarchy. This can be done simply by doing clock transformations (or calculating latencies at all stages in terms of absolute *nano* seconds). For convention, we refer to the DRAM and core clocks as t_{CLK}^{DRAM} and t_{CLK}^{CPU}, respectively.

- **Arbitration.** Arbiters are required in the system to regulate access to shared resources. The considered resources for arbitration are the request bus, the response bus, each bank of the LLC, the LLC-DRAM bus, and the three stages of the main memory (PRE, ACT, CAS). The DRAM-LLC bus does not require an arbiter as it does not incur any contention. This is because the DRAM-LLC bus is an on-chip bus between the DRAM memory controller and the LLC; therefore, its data transfer time is much lower than that of the off-chip DRAM access time. The arbitration scheme we propose to coordinate all these resources is discussed in detail in Section 4.1.

3.2 Latency Model

For any request r, let t_r^a be the time at which r arrives in the system and t_r^f the time at which r finishes executing. Formally, request r is outstanding in interval $[t_r^a, t_r^f]$. As discussed in Section 3.1, an OoO core can have multiple outstanding requests. Hence, it is essential to clarify how to compute the latency of a request. Using the same approach as in [31], we say that r is *oldest* at time t if it is the earliest arrived request of its core that is still outstanding at t. Note that because our architecture model allows multiple outstanding requests to execute in parallel and complete out-of-order, a request r might never become the oldest. However, if it does, it remains oldest until its finish time t_r^f. Furthermore, it must become oldest either at t_r^a, if there is no other outstanding request of the same core, or at the latest finish time of a request of the same core that arrived before r. The processing latency of a request is then the time during which it is oldest or zero if it never becomes oldest. Intuitively, this ensures that we do not count in the latency of a request the queuing delay caused by other requests of the same core that arrived before it. Note that when a core generates multiple concurrent requests, the time required to complete executing all requests is bounded by the sum of their processing latencies.

3.3 Request Model

In line with the delay composition theorem summarized in Section 2.3, we model each request r as executing on a sequence of stages corresponding to hardware resources in our system where requests are scheduled based on an arbiter. For LLC and interconnections, such resources comprise each of the N LLC banks, which we denote with BANK, the request bus REQ, the response bus RESP, and the LLC-DRAM bus SBUS. Note that the DRAM-LLC bus is not modeled as a pipeline stage since it is not subject to arbitration, as explained in Section 3.1.

Similar to [15, 16, 43], we model DRAM as consisting of three stages: PRE:ACT:CAS. Each stage models the interference of other requests on commands of the corresponding type. Note that because we assume private banks in DRAM, such interference can only be caused by intra-stage DRAM constraints. We define the execution time on any s stage of these stages as c_s. For instance, the execution time on the REQ stage is c_{REQ}, and on the SBUS stage is c_{SBUS}.

3.4 Request Types

Following the described request model, we classify requests into a set of request *types*; the type $\mathcal{T}(r)$ of request r determines the list of stages/resources traversed by r. We notice that requests are issued to the memory system for two main reasons: a load/store request for data that is miss in the L1 cache or a write-back request for a victim dirty cache line to the lower

memory level. Thus, the request types are split into two groups. The first one contains the demanding requests that are miss in the L1 cache, in which a core broadcasts a load/store request on the request bus and waits for the demanding data to be sent on the response bus ($\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ and $\mathcal{T}_4$). This is denoted in Figures 1b – 1e. On the other hand, Figures 1f – 1h show the write-back requests of a dirty evicted cache line from L1 and LLC, which compose the second group ($\mathcal{T}_5$, $\mathcal{T}_6$ and $\mathcal{T}_7$). We now explain each of the request types in detail.

$\mathcal{T}_1$) REQ:BANK:RESP. This type explains the path for a request demanding a cache line that exists in the LLC. After the core broadcasts a request on the request bus, the data is read from the LLC bank and then sent to the core on the response bus (Figure 1b).

$\mathcal{T}_2$) REQ:RESP/BANK. In this case, the request hits into data that is found in the LLC write buffer. Based on the miss forwarding optimization discussed in Section 3.1, the requested cache line will be sent to the core on the response bus while simultaneously being rewritten to the LLC bank (Figure 1c).

$\mathcal{T}_3$) REQ:SBUS:RESP/BANK. This represents the case of a request that misses in the LLC bank but hits in the MC. This happens when this request targets a cache line that has been recently evicted from the LLC and sent to the DRAM; and hence, not yet written into the DRAM banks. Therefore, the system will fetch the data directly from the memory controller, and send it back to be simultaneously processed on the corresponding LLC bank, while also being sent to the requesting core through the response bus similar to the previous type. This is depicted in Figure 1d.

$\mathcal{T}_4$) REQ:SBUS:PRE:ACT:CAS:RESP/BANK. This represents the case where the request needs to fetch the data from the DRAM device. Accordingly, once the request is issued in the request bus, and misses in the LLC, the request will be sent to the DRAM on the system bus. Afterwards, the data will be fetched from the DRAM bank through the PRE:ACT:CAS stages. Then, the fetched data will be sent to the LLC bank to be written to its data array and simultaneously to the requesting core through the response bus as illustrated in Figure 1e.

$\mathcal{T}_5$) REQ:RESP:BANK. This type corresponds to a write-back request from L1 to the LLC due to the eviction of a dirty line. It first sends a request on the request bus to notify the LLC that it is going to update a cache line, and then it puts the data on the response bus to the LLC. Finally, the LLC bank gets the data and processes it as delineated in Figure 1f.

$\mathcal{T}_6$) SBUS:PRE:ACT:CAS. This type represents the write-back from LLC to DRAM due to the eviction of a dirty up-to-date cache line from the LLC bank. The LLC sends concurrently an invalidation message to the L1 caches and a write-back request to the DRAM through the SBUS. Please note that as aforementioned, the back invalidation to the L1s happens in its dedicated bus, and hence, is not subject to arbitration. Therefore, it does not have a dedicated stage. In contrast, the write to the DRAM after traversing the SBUS, requires the three DRAM stages: PRE:ACT:CAS. This type is shown in Figure 1g.

$\mathcal{T}_7$) RESP:SBUS:PRE:ACT:CAS. Similar to $\mathcal{T}_6$, this type represents an LLC write-back of a dirty evicted line to the DRAM. However, unlike $\mathcal{T}_6$, the evicted line in this case is stale, which means that it is updated in the L1 cache. Thus, when the L1 cache receives the

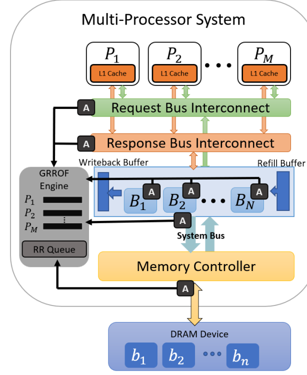


Figure 2 The architecture of GRROFproviding the global view for all arbiters.

invalidation message, it sends the updated version of the line to the LLC on the response bus. The cache controller stores the line in its write buffer until the data is received from L1 and then sends a write request to the main memory. Therefore, it requires RESP stage from L1 to the LLC, SBUS stage from LLC to the DRAM, and then the three DRAM stages: PRE:ACT:CAS. This type is shown in Figure 1h.

Two important general notes to make about the request types. First, requests of type $\mathcal{T}_2$, $\mathcal{T}_3$ and $\mathcal{T}_4$ execute in parallel on two stages ($BANK$ and $RESP$) instead of a linear sequence of stages which is the supported model by the existing pipelining analysis in [21]. To be able to pipeline, we apply the delay analysis to the two possible sequences and take the one that leads to the worst-case latency. More details on how to apply the delay analysis to such requests are in Section 5. Second, for the request types accessing the DRAM (namely, $\mathcal{T}_4$, $\mathcal{T}_6$ and $\mathcal{T}_7$), we assume a request targeting a closed row in the DRAM bank. This assumption is mandatory to provide safe worst-case latency bounds. This is because as explained in Section 2, requests targeting a closed DRAM row suffer larger delays compared to requests targeting an open row.

4 GRROF: Coordinating Management of All Memory Resource

In this section, we introduce the proposed architecture to coordinate all arbiters in the memory hierarchy, enabling us to apply the pipelining idea from the delay composition theorem. The high-level diagram of the proposed architecture is shown in Figure 2, and we use Figure 3 as a running example to explain its operation.

Since one of the motivations of this work is the inherent pessimism in considering each memory resource separately and then applying the additive latency approach, we start with an illustrative example that highlights this pessimism. Figure 3 considers a system with three cores $P_1 - P_3$ where each P_i issues three requests $r_{i,1}$-$r_{i,3}$ to the cache hierarchy. Request's arrival to the system is modeled by the up arrows $\uparrow$. For example, $r_{1,1}$, $r_{1,2}$, and $r_{1,3}$ from P_1 arrive at the timestamps 1, 9, and 17, respectively. The system has multiple arbitration stages REQ, BANK_{0-6}, and RESP. In Figure 3a, each stage employs an independent RR arbiter. Requests $r_{1,1}$, $r_{2,1}$, and $r_{3,1}$ target $BANK_0$, while the other requests are distributed over the other banks. According to the given scenario and the separate RR arbitration, request $r_{1,1}$ incurs a significant delay on RESP stage despite being the oldest request from P_1 and does not finish up until timestamp 58 (modeled by the down arrow $\downarrow$). More importantly,

if we use the additive latency approach naively without considering the fact that $r_{1,1}$ while being the oldest for P_1 might not be the local oldest at each separate resource, the bound will be the maximum possible interference due to requests from other cores in addition to the service time of $r_{1,1}$ itself in each resource. Since this is RR, it will be $2 \times 3 + 10 \times 3 + 5 \times 3 =$ 51 cycles. This is less than the actual suffered bound; and hence, is in fact an unsafe bound. The only possible way to derive a safe bound is to assume that the request under analysis is always arriving last to each resource after the maximum number of requests from the same core. As explained in Section 2.2, this provides a safe bound at the expense of being extremely pessimistic.

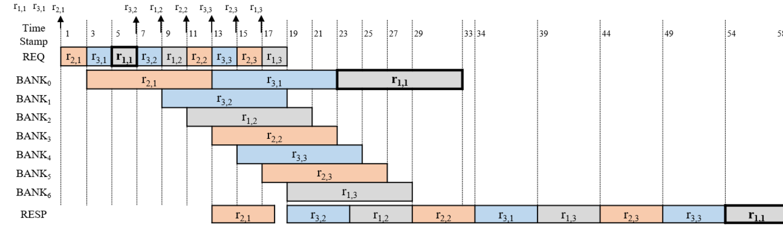
4.1 GRROF: Coordinated Management of All Memory Resources

Motivated by these observations about the existing direction in analyzing memory resources in multi-core real-time systems, we propose GRROF: a methodology to coordinate all the arbitration decisions across all resources in the memory hierarchy of a modern multi-core platform. This includes shared interconnects (request, response, and system buses), shared cache(s), and shared off-chip DRAM. The key idea behind this methodology is to enable all the arbiters in the memory hierarchy to use a shared state of the system to make a coordinated scheduling decision. It is important to emphasize that under GRROF, every resource in the memory hierarchy still deploys its own dedicated arbiter, which is essential for parallelism. This is in contrast to assuming a unified global arbiter that manages all the memory resources. We observe that, for instance, most of the existing works in cache analysis combine all interconnect resources to the shared cache as well as the shared cache itself into one resource that is arbitrated using one arbiter (e.g. [14, 24, 42]). Instead, in GRROF and as explained in Section 3, every resource has its own arbiter. So, there is a dedicated arbiter for the request bus, response bus, each LLC bank, system bus to the DRAM, and the DRAM memory controller. However, all these arbiters operate in coordination using a global view of the state of different requests in the system.

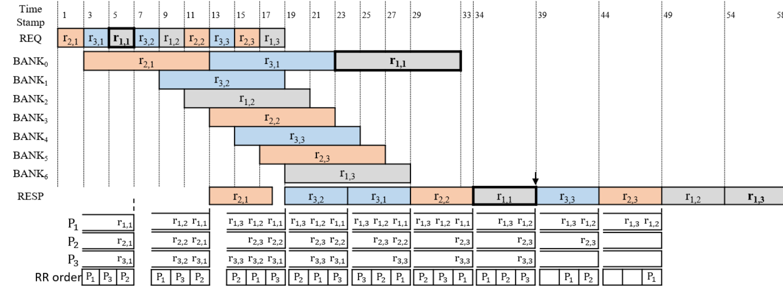
We now detail the operation of GRROF. The global view is maintained using the GRROF Engine in Figure 2. This engine maintains the following state.

1) RR Order. It maintains a RR order among the cores. Arbiters arbitrate among cores according to this RR order. Therefore, all arbiters see the same RR order view. A core gets pushed into the RR order queue if it has issued a request to the system (e.g., upon an L1 miss in our system model). Once a core is at the head of the RR order, it keeps that order until its oldest request retires from the system. For example, in Figure 3b, P_3 maintains its position at the top of the RR order despite having $r_{3,2}$ retired at timestamp 24. This is because the oldest request from P_3 is $r_{3,1}$, which still needs to finish. This is vital to provide tight guarantees for the oldest requests. *The intuition is that the oldest memory request is the one stalling the pipeline [34, 35]; and hence, contributing to the task's WCET [33].* In the same example in the figure, by keeping its RR position, P_3 manages to finish its oldest request $r_{3,1}$ at timestamp 29 compared to what happens for uncoordinated arbitration where P_3 loses its RR order in Figure 3a leading $r_{3,1}$ to wait for another slot for P_3 in the response bus resource and finishes at 39.

2) Per-Core FIFO Order. For arbiters to be able to determine the relative order of requests from the same core, the GRROF Engine maintains one First-In First-Out queue (FIFO) per core. Upon arrival to the system, a request ID is pushed in the corresponding core's FIFO. The request ID is removed from such FIFO upon retiring from the system. Accordingly,



(a) Uncoordinated RR arbitration.



(b) Proposed GRROF solution. The bottom of the figure shows the state tracked by the GRROF Engine, including the per-core FIFO order and the RR order among cores.

Figure 3 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, 10, and 5 cycles, respectively.

the FIFOs keep state about this relative order for requests from the same core. Again, all arbiters have access to these FIFOs and hence, can decide accordingly which request to elect for service at the arbitrated resource.

The arbiters deploy Round Robin Oldest First (RROF) arbitration [33]. They first conduct a RR among the oldest requests of the cores. Only if no older requests are ready to be serviced at that resource the arbiter conducts RR among younger requests (in the order of the FIFO for the same core and RR among cores). In the example in Figure 3b, the response bus (RESP) arbiter elects $r_{1,1}$ at timestamp 34 because it is the oldest request from the core at the top of the RR order. On the other hand, at 19, the RESP arbiter cannot issue any of the oldest requests ($r_{1,1}$, $r_{2,2}$, $r_{3,1}$) since none of them is ready for this resource. Accordingly, it picks $r_{3,2}$ as the only ready non-oldest request.

The important and novel aspect here is that this RROF at each resource uses the global system state from the GRROF Engine (Namely, RR Order and per-core request FIFO order). As a result, the relative request priorities remain the same for all arbitrated resources (stages in delay composition theorem terminology); hence, we can apply the pipelining from the theorem. We prove in our analysis in Section 5 how this enables us to significantly reduce the worst-case latencies suffered by the oldest requests in the system. Considering $r_{1,1}$ in Figure 3, we observe that $r_{1,1}$ arrives at the RESP bus resource last among all the requests since it has been delayed by $r_{2,1}$ and $r_{3,1}$ in the Bank_0 stage. As a result, in the uncoordinated RR baseline in Figure 3a, according to the local RR arbiter at the RESP stage, this request has to wait for all the requests to finish, including the non-oldest requests from the same core. We see in Figure 3a that $r_{1,1}$ suffers interference from requests from the two other cores in all the stages (REQ, BANK, and RESP) (Observation 1 in Section 2.2). Moreover, $r_{1,3}$ is serviced by the RESP before $r_{1,1}$ since locally, $r_{1,3}$ is considered older (Observation 2 in

Section 2.2). This overall leads $r_{1,1}$ to finish at timestamp 58. On the other hand, GRROF, in Figure 3b, is aware of the global order of all requests, thus once $r_{1,1}$ arrives at the RESP stage, it gets the highest priority among all P_1 requests since it is the oldest. Additionally, P_1 is at the top of the RR queue since its oldest request still needs to finish. As a result, $r_{1,1}$ is serviced at timestamp 39 instead of 58.

4.2 The Proposed Memory Controller

To be able to apply the pipelining from the delay composition theorem across the entire memory hierarchy, the schedulers inside the DRAM MC have to apply the exact arbitration mechanism described in Section 4.1. Therefore, we also propose an MC scheduler that utilizes the global request state from the GRROF Engine. In detail, we employ three distinct RROF arbiters for PRE, ACT, and CAS commands, which follow the global order maintained by the centralized GRROF Engine. If multiple commands from different arbiters are selected at the same time, a command bus conflict may happen. The MC handles this conflict by prioritizing CAS commands over ACT, and ACT over PRE commands, which is the common approach followed by COTS MCs. The intuition is that CAS commands are for row hit requests and hence prioritizing them will increase the overall system performance. However, the proposed MC strictly applies the RROF scheme between CAS commands and handles reads and writes equivalently. This is in contrast to DuoMC [32], which deploys read/write batching, where several read (write) commands are scheduled together as a batch and executed in a read (write) round, and rounds are alternating types. The reason for avoiding read/write batching is we find this to break the property required by the delay composition theorem. Basically, by read/write batching, requests no longer keep their relative priority across the different stages in the memory hierarchy. For example, a write might have a higher priority than a read in one of the cache levels (according to GRROF orders) but the write gets deprioritized at the MC (e.g. if it is executing a read batch upon its arrival to its request queues). Breaking this property hinders our goal of being able to apply pipelining to the memory latency analysis. Instead, by handling reads and writes in a similar fashion, we are able to apply the pipelining among DRAM command stages. This pipelining at the DRAM has been explored by previous works [15, 16, 43], albeit they considered DRAM only.

It is important to notice that deploying the same GRROF arbitration methodology at all memory resources including within cache hierarchy as well as in DRAM enables us to apply the delay composition theorem since all requests maintain their relative order throughout the resources. Without this coordination, 1) suffered delays at different stages cannot be overlapped by applying to the pipeline, and 2) the oldest request at one resource can become non-oldest at another resource and hence can be significantly delayed. This is similar to the clarified example in Figures 3a and 3b.

5 Latency Analysis

In this section, we show how to obtain a latency bound L_{ua} on the worst-case processing latency of the oldest request under analysis r_{ua} of a given type, following our described GRROF arbitration in Section 4.1 and MC in Section 4.2, and based on the delay composition framework.

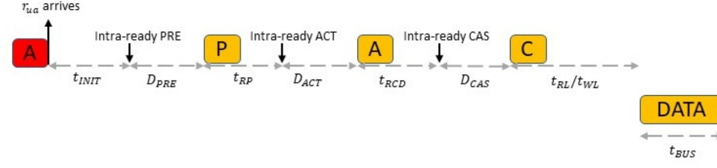


Figure 4 The delay composition for the request under analysis.

5.1 Main Memory Latency Analysis

Based on the proposed MC arbitration, we next determine an upper bound to the delay that the oldest request under analysis r_{ua} can suffer in the main memory. In the worst-case scenario, r_{ua} and all interfering requests require to issue three commands: PRE, ACT, and CAS. Figure 4 shows how to decompose the latency of r_{ua} based on two types of terms: 1) intra-bank constraints between commands issued to the same bank, or between command and data, and 2) inter-bank constraints caused by the same type of command issued to other banks. Since we assume private banks, intra-bank constraints can only be caused by commands of the core that issued r_{ua}, while requests of other cores cause inter-bank constraints. This section aims to derive the worst-case latency components of the three DRAM stages, which we will then use in Section 5 in order to calculate the total worst-case request latency. These components are as follows:

1. **For the PRE stage:** D_{PRE} is the maximum latency of PRE from the time it becomes intra-ready until it is issued, caused by interfering PRE commands of other cores.
2. **For the ACT stage:** D_{ACT} is the maximum latency of ACT from the time it becomes intra-ready until it is issued, caused by interfering ACT commands of other cores.
3. **For the CAS stage:** D_{CAS} is the maximum latency of CAS from when it becomes intra-ready until it is issued, again caused by interfering CAS commands of other cores.

We next derive bounds on $\mathcal{D}_{PRE}$, $\mathcal{D}_{ACT}$ and $\mathcal{D}_{CAS}$ as a function of the numbers N_{PRE}, N_{ACT} and N_{CAS} of higher priority requests whose PRE, ACT and CAS commands, respectively, interfere with the commands of r_{ua}. However, since, as aforementioned applying the pipelining from the delay composition theory to these three DRAM stages is not novel, and the latency components have already been derived in several previous works [15, 43], we do not formally prove their derivation and instead use the values directly from those works. That said, for comprehensiveness in the paper, we intuitively explain each equation.

For the PRE stage, Equation 1 calculates $\mathcal{D}_{PRE}$, which uses the same analysis as in [43] (Equation 2). The intuition behind Equation 1 is as follows. Since there is no inter-bank timing constraint between PRE commands, each interfering PRE contributes one clock cycle of delay; however, we have to add an additional cycle per command to account for the effects of command bus conflicts.

$$\mathcal{D}_{PRE}(N_{PRE}) = 2 \cdot N_{PRE} \cdot t_{clk}^{DRAM} \tag{1}$$

For the ACT stage, ACT commands have two inter-bank timing constraints, t^{l}_{RRD} which applies between successive commands, and t_{FAW} which applies every 4 commands. Hence, the bound must consider the maximum of the two constraints. The value of each timing constraint is increased by one clock cycle to account for bus conflicts caused by CAS commands. That is one difference between Equation 2 below and Equation 3 in [43]. In the latter, each timing constraint is instead increased by two clock cycles because ACT commands can suffer bus conflicts due to both PRE and CAS commands, while for our proposed controller, as

explained in Section 4.2, ACT will not suffer command bus contention from PRE. Finally, in Equation 2, $t_{FAW} - 3 \cdot t^{l}_{RRD} - t^{DRAM}_{clk}$ represents the maximum delay caused by ACT commands of lower-priority requests issued as late as possible before the ACT of r_{ua} becomes intra-ready.

$$\mathcal{D}_{ACT}(N_{ACT}) = t_{FAW} - 3 \cdot t^{l}_{RRD} - t^{DRAM}_{clk} + \max\Big(N_{ACT} \cdot (t^{l}_{RRD} + t^{DRAM}_{clk}),$$
$$\lfloor N_{ACT}/4 \rfloor \cdot (t_{FAW} + t^{DRAM}_{clk}) + (N_{ACT}\%4) \cdot (t^{l}_{RRD} + t^{DRAM}_{clk})\Big) \quad (2)$$

Finally, **for the CAS stage,** Equations 3 and 4 calculate $\mathcal{D}_{CAS}$ for read (RD) and write (WR) requests, respectively. Since in our controller, CAS commands have the highest priority for accessing the command bus; they cannot suffer command bus conflicts. Also, since in our controller RD and WR commands are scheduled fairly, contrary to [43] and similar to [15], the N_{CAS} interfering commands can comprise both RD and WR. Inter-bank timing constraints between CAS commands are longer when switching between RD-to-WR (t_{RTW}) and WR-to-RD ($t_{WL} + t_{Bus} + t_{WTR}$) compared to issuing two RD or two WR commands back-to-back (t^{l}_{CCD}); this is because the data bus needs this time to change the direction of the data sent on the bus. Hence, to bound the worst-case latency for a CAS command, we consider the maximum alternation between RD and WR commands. Again, a lower-priority CAS command can be issued one clock cycle before the CAS of r_{ua} becomes intra-ready; hence, the total number of interfering requests is $N_{CAS} + 1$. Noticing that the last constraint must be a WR-to-RD switch if r_{ua} is an RD (Equation 3), and an RD-to-WR switch if r_{ua} is a WR (Equation 4).

$$\mathcal{D}^{RD}_{CAS}(N_{CAS}) = \left\lfloor \frac{N_{CAS}+1}{2} \right\rfloor \cdot t_{RTW} + \left\lceil \frac{N_{CAS}+1}{2} \right\rceil \cdot (t_{WL} + t_{Bus} + t_{WTR}) - t^{DRAM}_{clk} \quad (3)$$

$$\mathcal{D}^{WR}_{CAS}(N_{CAS}) = \left\lceil \frac{N_{CAS}+1}{2} \right\rceil \cdot t_{RTW} + \left\lfloor \frac{N_{CAS}+1}{2} \right\rfloor \cdot (t_{WL} + t_{Bus} + t_{WTR}) - t^{DRAM}_{clk} \quad (4)$$

In addition to the latency of each of these stages, there are additional across-stage delays a request can suffer. These are as follows. 1) t_{INIT} is the worst-case latency from the time r_{ua} arrives at the MC to PRE becoming intra-ready. In the worst case depicted in the figure, a non-oldest request r of the same core as r_{ua} could issue an ACT command one cycle before r_{ua} arrives at the MC. In such a case, although r_{ua} preempts r, the ACT command imposes an ACT-to-PRE timing constraint t_{RAS}. Thus, in the worst case we have $t_{INIT} = t_{RAS} - t^{DRAM}_{clk}$. Additionally, intra-bank constraints impact when a request can become ready at a particular stage. Namely, 2) t_{RP} is the PRE-to-ACT timing constraint, and 3) t_{RCD} is the ACT-to-CAS timing constraint. We now show how to use all these components to derive the total end-to-end worst-case latency of a memory request.

5.2 Holistic Memory Latency Bound

To maximize L_{ua}, we assume that r_{ua} becomes oldest at the earliest possible time $t^{a}_{r_{ua}}$. Since we have several request types for r_{ua} as well as for interfering requests. Proving the worst-case L_{ua} for all scenarios is impossible within the paper space. Instead, we developed a brute-force algorithm that covers all possible scenarios and calculates their latency to ensure that we correctly compute the latency bound for every request type and values of timing parameters and corresponding valid values of $N_{REQ}, N_{SBUS}, N_{PRE}, N_{ACT}, N_{CAS}, N_{RESP}, N_{BANK}$[1]. It

[1] source code is available here: `https://gitlab.com/FanosLab/endtoend_wcl_cases_matlab/`

is worth noting that the running time of this brute-force machinery is in the range of few seconds. That said, it is done offline; therefore, the exact run-time complexity is irrelevant to the proposed solution. In the rest of this section, we focus on deriving the global worst-case L_{ua} across all scenarios, which corresponds to a r_{ua} of type $\mathcal{T}_4$, which traverses the most stages and has the highest latency for our system. First note that as pointed out in Section 3.4, requests of type $\mathcal{T}_4$ (as well as those of type $\mathcal{T}_2, \mathcal{T}_3$) execute simultaneously on BANK and RESP, rather than traversing a linear sequence of stages. Since BANK and RESP are the last stages on which r_{ua} executes, its latency depends on which of the two stages it last finishes execution on. If r_{ua} finishes executing on BANK after RESP, then we can obtain a latency bound by analyzing its execution along path (REQ, SBUS, PRE, ACT, CAS, BANK); otherwise, by analyzing path (REQ, SBUS, PRE, ACT, CAS, RESP). Therefore, we can compute L_{ua} by applying the delay composition analysis to both stage sequences and taking the maximum obtained bound.

Second, as discussed in Section 2.3, we use the improved delay composition analysis in [22] to support requests of different types. This analysis's key idea is to split every higher priority interfering request r into a set of *segments*: each segment represents the execution of r on a sequence of consecutive stages in the path of the request under analysis encountered either in the same or exactly in reverse order. We consider the path (REQ, SBUS, PRE, ACT, CAS, RESP) as an example since we find it to lead to the maximum possible L_{ua} in our system. Then, the following segments must be considered:

- Each request of type $\mathcal{T}_1, \mathcal{T}_2$ or $\mathcal{T}_5$ is split into a segment (REQ) and a segment (RESP). Note that no segment can represent execution on BANK since this stage is not part of the analyzed path of r_{ua}.
- Each request of type $\mathcal{T}_3$ is split into a segment (REQ, SBUS) and a segment (RESP).
- Each request of type $\mathcal{T}_4$ corresponds to a single segment (REQ, SBUS, PRE, ACT, CAS, RESP).
- Each request of type $\mathcal{T}_6$ corresponds to a single segment (SBUS, PRE, ACT, CAS).
- Each request of type $\mathcal{T}_7$ is split into a segment (RESP) and a segment (SBUS, PRE, ACT, CAS).

L_{ua} is obtained by summing the following terms:

1. L_{ua}^{trav}: this is the time required by r_{ua} to traverse its required stages (based on type), assuming it suffers no interference at all. For each stage, this is the maximum time required to move to the next one along the path or the time needed to finish executing the last stage.
2. L_{ua}^{lp}: this is the latency component caused by low-priority requests. For each stage, the maximum interference is caused by a single lower-priority request. In GRROF, such request must start executing no later than one clock cycle before r_{ua} or any higher-priority request becomes ready at that stage, otherwise the arbiter will not select the lower-priority request; therefore, the maximum interference is equal to the execution time of any lower-priority request on that stage minus one clock cycle.
3. L_{ua}^{hp}: this is the latency component caused by higher-priority requests. For every segment, its maximum execution time on any one stage on which it executes.

We begin by discussing L_{ua}^{trav}. Here, we are interested in the time between a request starting execution on a stage and becoming ready to be arbitrated on the next stage. Since the request becomes ready on SBUS immediately after finishing executing on REQ, the time from REQ to SBUS is simply the execution time c_{REQ} on REQ. Similarly, the time to finish executing on RESP is c_{RESP}. However, in the case of DRAM stages, we have to consider the

effect of intra-bank DRAM constraints. As discussed in Section 5.1, the time from PRE to ACT is t_{RP} and the time from ACT to CAS is t_{RCD}. For the time from SBUS to PRE, we have to consider three time components: (a) the time c_{SBUS} required to execute on SBUS; (b) the time t_{cross} required to cross clock domains since the system bus and the DRAM controller use different clocks. In the worst case, we assume that such time equals one clock cycle of the destination domain, i.e., $t_{cross} = 1$. (c) The time $t_{INIT} = t_{RAS} - 1$ required for the request to become ready on PRE after arriving at the memory controller. Hence, the required time is equal to $c_{SBUS} + t_{RAS}$. Finally, for CAS to RESP, the time includes $t_{RL} + t_{BUS}$ to obtain the data after issuing the CAS command, $t_{cross} = 1$ to cross back into the CPU clock domain [2], and c_{SBUS} to send the data back through the DRAM-LLC bus, for a total of $t_{RL} + t_{BUS} + 1 + c_{SBUS}$. Summing over all stages, for our example, we obtain a total of:

$$\begin{aligned} L_{ua}^{trav} =& c_{REQ} + c_{SBUS} + 1 + \\ & \big((t_{RAS} - 1) + t_{RP} + t_{RCD} + t_{RL} + t_{BUS}\big) \cdot (t_{CLK}^{DRAM}/t_{CLK}^{CPU}) + c_{SBUS} + c_{RESP}. \end{aligned} \tag{5}$$

Next, we consider the interference of lower and higher-priority requests/segments ($L_{ua}^{intf} = L_{ua}^{hp} + L_{ua}^{lp}$). Let $N_{REQ}, N_{SBUS}, N_{PRE}$, N_{ACT}, N_{CAS} to denote the number of segments that interfere on the corresponding stage, subject to the constraint that each segment interferes on only one stage. Then including the effect of a lower-priority request, the total interference on REQ, SBUS and RESP is equal to $\mathcal{D}_{REQ}(N_{REQ}) = c_{REQ} - 1 + N_{REQ} \cdot c_{REQ}$, $\mathcal{D}_{SBUS}(N_{SBUS}) = c_{SBUS} - 1 + N_{SBUS} \cdot c_{SBUS}$, $\mathcal{D}_{RESP}(N_{RESP}) = c_{RESP} - 1 + N_{RESP} \cdot c_{RESP}$; while the interference on PRE, ACT and CAS is equal to $\mathcal{D}_{PRE}(N_{PRE}), \mathcal{D}_{ACT}(N_{ACT})$ and $\mathcal{D}_{CAS}^{RD}(N_{CAS})$ as computed in Equations 1, 2, 3. The total interference is thus calculated by Equation 6 maximized over all possible values of $N_{REQ}, N_{SBUS}, N_{PRE}, N_{ACT}, N_{CAS}, N_{RESP}$.

$$\begin{aligned} L_{ua}^{intf} =& \mathcal{D}_{REQ}(N_{REQ}) + \mathcal{D}_{SBUS}(N_{SBUS}) + \mathcal{D}_{RESP}(N_{RESP}) + \\ & \big(\mathcal{D}_{PRE}(N_{PRE}) + \mathcal{D}_{ACT}(N_{ACT}) + \mathcal{D}_{CAS}^{RD}(N_{CAS})\big) \cdot (t_{CLK}^{DRAM}/t_{CLK}^{CPU}) \end{aligned} \tag{6}$$

Note that based on our GRROF arbitration, at most $M-1$ requests can have higher priority than r_{ua}. For the system settings employed in our evaluation, Equation 6 is maximized when all $M-1$ requests are of type $\mathcal{T}_7$, yielding $M-1$ segments of type (RESP) with $N_{RESP} = M-1$ and $M-1$ segments of type (SBUS, PRE, ACT, CAS) interfering on CAS (i.e., $N_{CAS} = M-1$), for a resulting interference:

$$\begin{aligned} L_{ua}^{intf,GRROF} =& \mathcal{D}_{REQ}(0) + \mathcal{D}_{SBUS}(0) + + \mathcal{D}_{RESP}(M-1) + \\ & \big(\mathcal{D}_{PRE}(0) + \mathcal{D}_{ACT}(0) + \mathcal{D}_{CAS}^{RD}(M-1)\big) \cdot (t_{CLK}^{DRAM}/t_{CLK}^{CPU}). \end{aligned} \tag{7}$$

Summing Equations 5 and 7 then yields a latency bound for path (REQ, SBUS, PRE, ACT, CAS, RESP), which again based on our system setting, is the latency bound L_{ua} for requests of type $\mathcal{T}_4$ (Equation 8).

$$L_{ua}^{GRROF} = L_{ua}^{trav} + L_{ua}^{intf,GRROF} \tag{8}$$

[2] Note that for the system in Section 6, we assume that the CPU clock has double the frequency of the DRAM clock and that the two clocks are synchronized. Under such assumption, we can take $t_{cross} = 0$.

Finally, note that when analyzing a path through BANK, the maximum interference is similarly equal to $\mathcal{D}_{BANK}(N_{BANK}) = c_{BANK} - t_{CLK}^{CPU} + N_{BANK} \cdot c_{BANK}$. It is important to consider that here N_{BANK} represents the number of segments that interfere on the *same* LLC bank as r_{ua}. However, a higher-priority request might also target a *different* LLC bank. For example, when $\mathcal{T}(r_{ua}) = \mathcal{T}_1$, a higher priority request r also of type $\mathcal{T}_1$ would yield a single segment (REQ, BANK, RESP) if it targets the same bank as r_{ua}, and two segments (REQ) and (RESP) if it targets a different bank. If $c_{BANK} \geq c_{REQ} + c_{RESP}$, then the first case results in higher interference; otherwise, the second.

5.3 Effect of Additive Latency Approach

After deriving the latency bounds for GRROF and using the pipelining analysis from the delay composition theorem, this section shows how pessimistic the resulting bounds are upon considering resources separately and follow the additive latency approach even when pipelining has been considered at one or more of the components in the system but not at the holistic level of the memory. In doing so, we first derive this bound for two systems that apply the latency additive theorem at different scale. Please note that to derive a safe bound using this approach, we follow the direction discussed in Section 2.2 by assuming the maximum possible delay from younger requests of the same core at each resource. As explained in Section 2.2, this provides a safe bound at the expense of being extremely pessimistic. We first define the two systems as follows. Discrete-RR is a system that deploys traditional RR arbitration at the REQ, RESP, and BANK stages, while it uses the MC model proposed in Section 4.2 deploying the RROF arbitration at the three DRAM stages: PRE, ACT, and CAS. Split-RROF is a system that deploys RROF arbitration locally at DRAM stages using the MC model proposed in Section 4.2 and at the cache stages. However, there is no coordination between the DRAM subsystem and the cache subsystem stages. We use a DRAM with a pipelined stages model for the two systems since the state-of-the-art analysis in DRAM already applies the delay composition theorem [15,16,43]. The Discrete-RR system, on the other hand, is using a traditional RR arbiter at each of the remaining three stages: REQ, BANK, and RESP. In contrast, the Split-GRROF goes one step further and even pipelines these three stages together. The reason for choosing this model is to show that even when pipelining resources at one of the levels and not at the system level, latency bounds are still quite pessimistic.

The latency bound L_{ua} for Discrete-RR, L_{ua}^{DRR} can be calculated as follows. First, the latency of each of the REQ, RESP, BANK, and SBUS stages has to be separately calculated. Since each of these stages adopts a RR arbiter and each core has a maximum of N_{pend} pending requests. The worst-case for r_{ua} is to assume that it arrives at the stage after $N_{pend} - 1$ requests from P_{ua} and that P_{ua} is last in the RR order. This yields a total of $M \cdot N_{pend} \cdot c_s$, where s is any of the REQ, RESP, BANK, or SBUS stages, which includes the execution of r_{ua} itself in s. For the DRAM stages, there is one clock cycle for domain crossing. Afterwards, in contrast to GRROF since the relative priority of requests can no longer be assumed to be the same between the cache and the DRAM subsystems, r_{ua} has to assume that it arrives at the DRAM in worst-case as the last request similar to all other stages. Therefore, it has to wait for $N_{pend}-1$ requests to finish from P_{ua}, which in worst-case are all write requests. This is the second line in Equation 9. Afterwards, r_{ua} itself can suffer $M - 1$ requests from other cores due to RR order. This is the third line in Equation 9. Since DRAM is pipelined, we maximized the delay over the ACT, PRE, and CAS stages, which

happens to be that in worst-case all the $M-1$ interfering requests are contributing to the CAS stage similar to GRROF in Equation 7. Applying the additive latency theorem, this gives a total delay for the Discrete-RR as follows:

$$L_{ua}^{DRR} = M \cdot N_{pend} \cdot (c_{REQ} + c_{SBUS} + c_{BANK}) + 1 + c_{SBUS}$$
$$\big((N_{pend}-1)(\mathcal{D}_{PRE}(0) + t_{RP} + \mathcal{D}_{ACT}(0) + t_{RCD} + \mathcal{D}_{CAS}^{WR}(M-1) + t_{WL} + t_{BUS} + t_{WR}) +$$
$$\mathcal{D}_{PRE}(0) + \mathcal{D}_{ACT}(0) + \mathcal{D}_{CAS}^{RD}(M-1) + t_{RL} + t_{BUS}\big) \cdot (t_{CLK}^{DRAM}/t_{CLK}^{CPU}). \quad (9)$$

The latency bound L_{ua} for Split-RROF, L_{ua}^{SRROF} can be calculated as in Equation 10. One important observation to highlight is that because of the pipelining effect in the cache subsystem, Split-RROF has to account only for interference from $M-1$ requests instead of the $M \cdot N_{pend}$ in Discerete-RR case. However, because Split-RROF does not pipeline the cache and the DRAM subsystems together, a special consideration has to be paid for what requests interfere with r_{ua} in the cache subsystem. In particular, because there are now requests that will hit in the BANK and requests that miss and go to the DRAM, requests will take different paths in the cache pipeline stage. The hit requests will be REQ, BANK, and RESP, while the miss requests (from the cache pipeline perspective) will be REQ, then go off the system (to DRAM), then come back to execute RESP and BANK in parallel. Due to this fact, r_{ua} can indeed suffer interference both at the REQ stage and at the BANK stage. This is accounted for in the second line in Equation 10. The first line represents the traversing latency similar to Equation 5 for GRROF. It basically goes through REQ, SBUS, then cross to DRAM (one cycle for clock domain crossing), and then comes back from DRAM through SBUS and then is processed in BANK. The third and fourth lines are accounting for DRAM interference very similar to Equation 9. The last line accounts for one low-priority request at each stage r_{ua} traverses.

$$L_{ua}^{SRROF} = c_{REQ} + c_{SBUS} + 1 + c_{SBUS} + c_{BANK} +$$
$$\mathcal{D}_{REQ}(M-1) + \mathcal{D}_{SBUS}(0) + \mathcal{D}_{BANK}(M-1) +$$
$$\big((N_{pend}-1)(\mathcal{D}_{PRE}(0) + t_{RP} + \mathcal{D}_{ACT}(0) + t_{RCD} + \mathcal{D}_{CAS}^{WR}(M-1) + t_{WL} + t_{BUS} + t_{WR}) +$$
$$\mathcal{D}_{PRE}(0) + \mathcal{D}_{ACT}(0) + \mathcal{D}_{CAS}^{RD}(M-1) + t_{RL} + t_{BUS}\big) \cdot (t_{CLK}^{DRAM}/t_{CLK}^{CPU}) +$$
$$(c_{REQ}-1) + (c_{SBUS}-1) + (c_{BANK}-1) \quad (10)$$

6 Evaluation Results

We implement the proposed solution as well as the two systems we compare against (Discrete-RR and Split-RROF from Section 5.3) on a cycle-accurate simulation platform integrating the cache subsystem simulator provided in [17] with MCsim as a main memory system simulator [29], in order to mimic the whole path of a request. By this way, we are able to accurately obtain end-to-end latency for memory accesses.

Experimental Setup. Unless otherwise specified, in all our experiments we use a quad-core system clocked at 2.4 GHz, where each core is OoO with up to 16 N_{pend} and a 32 KB 4-way set-associative private L1 cache. Through a split-transaction interconnect, the cores share access to a 4 MB 8-ways set-associative bankized LLC that comprises 8 separate banks. Both L1s and the LLC are write-back write-allocate non-blocking caches with a cache line size of 64 bytes and an LRU replacement policy. The LLC is set-partitioned such that each core has its own private sets. Nonetheless, all cores can access all LLC banks. To map the requests to the different LLC banks, the cache controller uses the Least-Significant-Bits (LSBs) in the

tag address bits to denote the bank number, such that a core can access all banks within its private sets. We use a DDR4-2400U for the main memory with a single-channel single-rank DRAM device. DRAM banks are partitioned such that each core accesses its own private set of banks. We assume that processing L1 hit requests takes a single cycle and processing data in the LLC data array takes ten cycles ($c_{BANK} = 10$). We also configure processing time on request and response buses to 2 and 5 cycles, respectively ($t_{REQ} = 2$, $t_{RESP} = 5$). The system bus between the LLC and the MC has a latency of ($t_{SBUS} = 5$) in either direction LLC-DRAM or DRAM-LLC.

Workloads. We use SPEC CPU benchmark [40]. While running a SPEC workload on one of the cores, the other cores are running a stressing microbenchmark to generate the most interference on the task under analysis. For these stressors, we use the *latency* benchmark from the IsolBench suite [41].

Compared Systems. In addition to the two real-time systems of (Discrete-RR and Split-RROF, we compare GRROF against COTS high performance arbiter using FRFCFS arbitration for all resources.

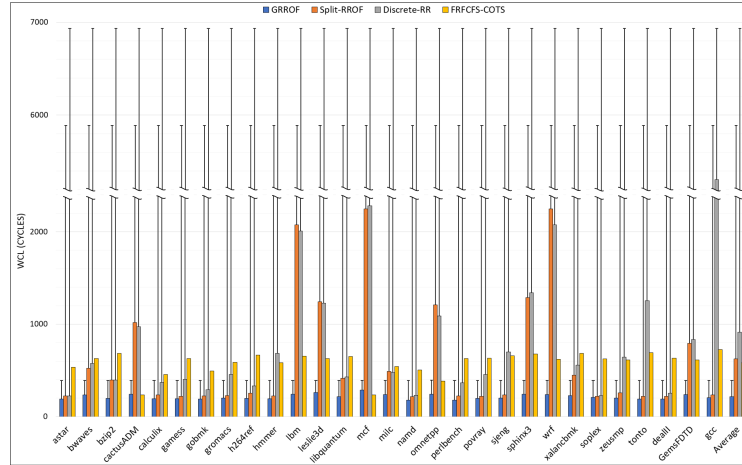


Figure 5 Per-request worst-case latency both Observed (solid bars) and analytical (T-shape) for SPEC workloads.

6.1 Per-Request Worst-Case Latency

Figure 5 delineates both the observed and analytical WCL suffered by any memory request for all the systems. Note that the high performance FRFCFS is theoretically unbounded (assuming that there is no threshold for requests reordering), thus there is no estimated bound for it. The figure shows that: 1) the analytical bounds for Split-RROF and Discrete-RR are very pessimistic. Compared to the observed WCL, they reach up to 26× (*namd*) and 30× (*astar*) for Split-RROF and Discrete-RR, respectively. This clearly shows the pessimism of the additive latency approach, as discussed throughout the paper. 2) GRROF manifests the lowest observed WCL per-request for all the workloads. For the worst-case observed latency across all the workloads, GRROF shows 8× and 18.4× reduction compared to Split-RROF and Discrete-RR, respectively. 3) The analytical bound of GRROF is the tightest latency bound which does not exceed than 2× of the experimental latency (*namd*). In fact, GRROF

achieves a 14.8 and 17.5 reduction on the analytical latency bound compared to Split-RROF and Discrete-RR, respectively. This experiment clearly emphasizes that using conventional per-resource real-time arbitration schemes alongside the additive latency approach suffers from excessively pessimistic latency bounds and that coordinating these resources such that pipelining analysis can be applied has a huge potential as an alternative.

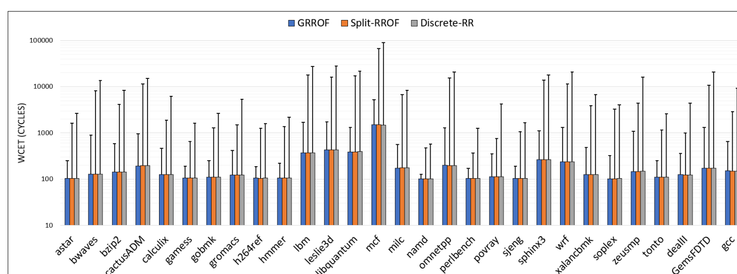


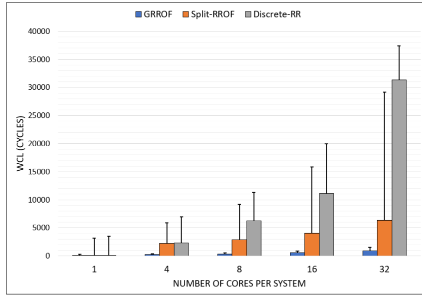
Figure 6 Per-task observed memory latency for SPEC benchmarks (solid bars), compared to the analytical memory processing time (T-shape). Values in y-axis are in logarithmic scale.

6.2 Per-Task Worst-Case Memory Latency

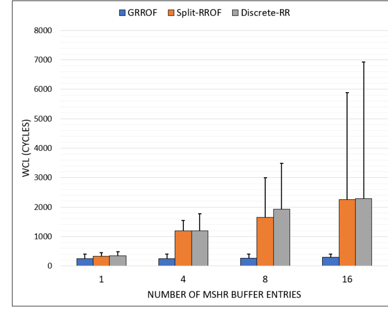
In this experiment, we evaluate the total task Memory Latency and compute the analytical total task's worst-case memory latency (WCML). Figure 6 shows the experimental total memory latency for SPEC benchmarks. Additionally, it shows the analytical WCML as T-bar. The analytical bound for each task is obtained by summing up the following components: 1) the number of L1 hit requests multiplied by the L1 hit latency, 2) the number of LLC hit requests multiplied by the WCL of a request hitting in LLC (this is type $\mathcal{T}_1$ in this case), and 3) the number of DRAM access requests multiplied by WCL of a miss (this is type $\mathcal{T}_4$ as driven in Equation 8). Please note that write-backs are not considered in this task analysis since they are neither stalling core pipeline nor in the critical path of the requests based on the considered system architecture in Section 3.1. From the figure, we observe that, the observed total memory time for the three systems are very close for the SPEC benchmarks. When investigating the reason for this, we found that most of the SPEC BMs exhibit a very high L1 hit rate. However and more importantly from a real-time prespective, in terms of predictability, the calculated bounds for Split-RROF and Discrete-RR are drastically pessimistic. In case of Split-RROF, the analytical WCML varies between 2×-36× of the observed latency. And for Discrete-RR, it varies between 4×-48×. This wide variability makes them poor in predictability and entails bounds not very useful. By looking at GRROF bounds, on the other hand, it is clear that it provides the tightest WCML, which does not overrun 1.5× and can be as close as 16% of the actual experimental latency.

6.3 Sensitivity Test

In this section, we conduct two experiments that study the request worst-case latency while: 1) increasing number of cores in the system (Figure 7a), and 2) increasing the size of MSHR buffer entries (and hence, the $N_{pending}$ from each core) (7b). In both figures, we show the results for only one SPEC benchmark (*mcf*). We observe similar trend for all the other benchmarks. For the first experiment, we experiment with 1, 4, 8, 16 and 32 core systems. Figure 7a emphasizes that the bound of GRROF increases linearly with number of cores. Likewise the previous experiment, we run SPEC workload aside with a stressing workloads. It



(a) Increasing number of cores.



(b) Increasing N_{pend}.

Figure 7 Per-request worst-case latency of *mcf* workload from SPEC benchmark. Column and T-shape bars denote experimental and analytical values, respectively.

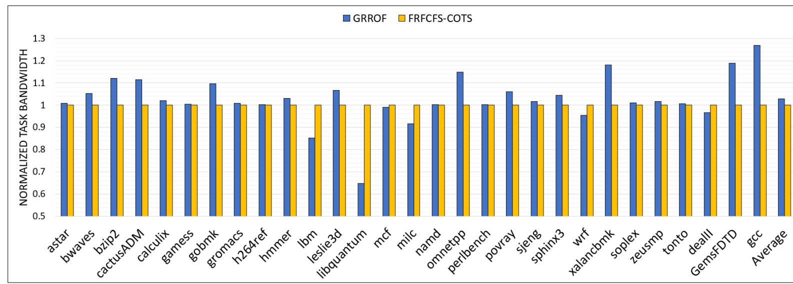


Figure 8 The bandwidth of SPEC workloads in a quad-core system.

is clear that GRROF's estimated latency is tightly bounded and very close to the experimental results, which does not exceed 0.6× of the observed WCL, in the most interfering setup of 32-cores. However, Split-RROF and Discrete-RR systems shows very large execution time which increases dramatically with increasing number of interfering cores, up to 8× and 32× of the observed WCL of GRROF. For the second experiment, it conveys the results for increasing number of N_{pend} entries as 1, 4, 8 and 16. In Figure 7b, the latency for GRROF is fixed and independent of the number of the N_{pend} entries. This is because, regardless of N_{pend}, GRROF ensures that the latency of any request can suffer interference delays from only one request from every other core as shown in Section 5. On the other hand, although Discrete-RR can provide a bound on the WCL for memory accesses, it is quadratically increased by the increasing number of outstanding requests on OoO systems. Requests may suffer up to 165× of their latency on an IO system. Split-RROF reduces this large variance, however the requests can suffer up to 12× of their latency on an IO system.

6.4 Average Performance

In this experiment, we evaluate the average-performance of GRROF. Figure 8 shows the average memory bandwidth of SPEC benchmarks running on GRROF and FRFCFS-COTS, and normalized on FRFCFS-COTS performance. Comparing the average-performance with the high-performance system, we make these observation points: 1) the memory bandwidth of GRROF is on-par with the COTS solution and performs even slightly better (2.9%) on average results. The reason for this improvement is that GRROF introduces more fairness to all benchmarks by prioritizing the oldest requests from all cores over younger ones. This protects tasks from severe interference from other co-running aggressor tasks.

7 Conclusions

In this paper, we introduce a coordinating management mechanism for the holistic memory system in order to sustain priorities of requests across all the shared resources in the memory system. By virtue of this novel mechanism, we could tightly bound the estimated per-request and per-task memory latency. And by comparing the proposed solution to the conventional real-time solutions, we made the point their analysis model is not convenient nor reliable for tightly bounding the latencies. In addition, we show that our system drastically reduces the WCL with more than 18× reduction in memory latency and tightly bounds the estimated latency by not exceeding 16% of the experimental latency.

References

1. Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: A predictable sdram memory controller. In *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256, 2007. `doi:10.1145/1289816.1289877`.
2. Intel® iris® plus graphics and uhd graphics open source. programmer's reference manualintel® 64 and ia-32 architectures optimization reference manual. `https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-icllp-vol07-memory_cache_0.pdf`.
3. ARM. Arm926ej-s™ revision: r0p5 technical reference manual. `https://developer.arm.com/documentation/ddi0198/e`, 2008.
4. ARM. Cortex-m4 technical reference manual r0p0. `https://developer.arm.com/documentation/ddi0439/b`, 2010.
5. Michael G Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 357–367, 2019. `doi:10.1109/RTAS.2019.00037`.
6. Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf 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 (RTCSA)*, pages 1–10, 2014. `doi:10.1109/RTCSA.2014.6910550`.
7. Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–15, 2013. `doi:10.1109/EMSOFT.2013.6658595`.
8. G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2), 2015. `doi:10.1145/2830555`.
9. Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A comparative study of predictable DRAM controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 2018. `doi:10.1145/3158208`.
10. Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS)*, pages 299–308, 2016. `doi:10.1145/2997465.2997471`.
11. Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *2009 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 68–77, 2009. `doi:10.1109/RTSS.2009.34`.
12. Mohamed Hassan. Heterogeneous mpsocs for mixed-criticality systems: Challenges and opportunities. *IEEE Design & Test*, pages 47–55, 2017. `doi:10.1109/MDAT.2017.2771447`.

13 Mohamed Hassan. Disco: Time-compositional cache coherence for multi-core real-time embedded systems. *IEEE Transactions on Computers (TC)*, pages 1163–1177, 2022. doi:10.1109/TC.2022.3193624.

14 Mohamed Hassan and Hiren 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)*, pages 1–11, 2016. doi:10.1109/RTAS.2016.7461327.

15 Mohamed Hassan and Rodolfo Pellizzoni. Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 2323–2336, 2018. doi:10.1109/TCAD.2018.2857379.

16 Mohamed Hassan and Rodolfo Pellizzoni. Analysis of memory contention in heterogeneous cots mpsocs. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 23:1–23:24, 2020. doi:10.4230/LIPIcs.ECRTS.2020.23.

17 Salah Hessien and Mohamed 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)*, pages 218–230, 2020. doi:10.1109/RTSS49844.2020.00029.

18 Mohamed Hossam and Mohamed Hassan. Predictably and efficiently integrating cots cache coherence in real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 17:1–17:23, 2022. doi:10.4230/LIPIcs.ECRTS.2022.17.

19 Intel. Write combining memory implementation guidelines. https://download.intel.com/design/PentiumII/applnots/24442201.pdf, 1998.

20 Javier Jalle, Eduardo Quinones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In *2014 IEEE Real-Time Systems Symposium (RTSS)*, pages 207–217, 2014. doi:10.1109/RTSS.2014.23.

21 Praveen Jayachandran and Tarek Abdelzaher. Delay composition in preemptive and non-preemptive real-time pipelines. *Real-Time Systems*, pages 290–320, 2008. doi:10.1007/s11241-008-9056-3.

22 Praveen Jayachandran and Tarek F. Abdelzaher. End-to-end delay analysis of distributed systems with cycles in the task graph. In *In Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 13–22, 2009. doi:10.1109/ECRTS.2009.15.

23 Anirudh M. Kaushik, Mohamed Hassan, and Hiren Patel. Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems. *IEEE Transactons on Computers (TC)*, pages 1–23, 2020. doi:10.1109/TC.2020.3037747.

24 Anirudh Mohan Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren Patel. Carp: A data communication mechanism for multi-core mixed-criticality systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 419–432, 2019. doi:10.1109/RTSS46320.2019.00044.

25 Ondrej Kotaba, Jan Nowotsch, Michael Paulitsch, Stefan M. Petters, and Henrik Theiling. Multicore in real-time systems – temporal isolation challenges due to shared resources. In *Design, Automation and Test in Europe (DATE)*, 2013.

26 David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA)*, pages 81–87, 1981.

27 NG Chetan Kumar, Sudhanshu Vyas, Ron K Cytron, Christopher D Gill, Joseph Zambreno, and Phillip H Jones. Cache design for mixed criticality real-time systems. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 513–516, 2014. doi:10.1109/ICCD.2014.6974730.

28 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, 2013. doi:10.1109/RTAS.2013.6531078.

29 Reza Mirosanlou, Danlu Guo, Mohamed Hassan, and Rodolfo Pellizzoni. Mcsim: An extensible dram memory controller simulator. *IEEE Computer Architecture Letters (LCA)*, pages 105–109, 2020. doi:10.1109/LCA.2020.3008288.

30 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Drambulism: Balancing performance and predictability through dynamic pipelining. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 82–94, 2020. doi:10.1109/RTAS48715.2020.00-15.

31 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Duetto: Latency guarantees at minimal performance cost. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1136–1141, 2021. doi:10.23919/DATE51398.2021.9474062.

32 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. DuoMC: Tight DRAM Latency Bounds with Shared Banks and Near-COTS Performance. In *ACM International Symposium on Memory Systems (MEMSYS)*, pages 1–14, 2021. doi:10.1145/3488423.3519322.

33 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Parallelism-Aware High-Performance Cache Coherence with Tight Latency Bounds. In *34th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 16:1–16:27, 2022. doi:10.4230/LIPIcs.ECRTS.2022.16.

34 Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 146–160, 2007. doi:10.1109/MICRO.2007.21.

35 Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ACM SIGARCH Computer Architecture News (ISCA)*, pages 63–74, 2008. doi:10.1109/ISCA.2008.7.

36 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 269–279, 2011. doi:10.1109/RTAS.2011.33.

37 Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. Towards time-predictable data caches for chip-multiprocessors. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems (SEUS)*, pages 180–191, 2009. doi:10.1007/978-3-642-10265-3_17.

38 DDR4 SDRAM Standard, JEDEC JESD79-4, 2012.

39 John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

40 Sarabjeet Singh and Manu Awasthi. Memory centric characterization and analysis of spec cpu2017 suite. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 285–292, 2019. doi:10.1145/3297663.3310311.

41 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016. doi:10.1109/RTAS.2016.7461361.

42 Man-Ki Yoon, Jung-Eun Kim, and Lui Sha. Optimizing tunable wcet with shared resource allocation and arbitration in hard real-time multicore systems. In *2011 IEEE 32nd Real-Time Systems Symposium (RTSS)*, pages 227–238, 2011. doi:10.1109/RTSS.2011.28.

43 Heechul Yun, Rodolfo Pellizzoni, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 184–195, 2015. doi:10.1109/ECRTS.2015.24.