DuoMC: Tight DRAM Latency Bounds with Shared Banks and Near-COTS Performance

Reza Mirosanlou
University of Waterloo
Waterloo, Canada
rmirosan@uwaterloo.ca

Mohamed Hassan
McMaster University
Hamilton, Canada
mohamed.hassan@mcmaster.ca

Rodolfo Pellizzoni
University of Waterloo
Waterloo, Canada
rpellizz@uwaterloo.ca

ABSTRACT

DRAM memory controllers (MCs) in COTS systems are designed primarily for average performance, offering no worst-case guarantees, while real-time MCs provide timing guarantees at the cost of a significant average performance degradation. For this reason, hardware vendors have been reluctant to integrate real-time solutions in high-performance platforms. In this paper, we overcome this performance-predictability trade-off by introducing DuoMC, a novel memory controller that promotes to augment COTS MCs with a real-time scheduler and run-time monitoring to provide predictability guarantees. Leveraging the fact that the resource is barely overloaded, DuoMC allows the system to enjoy the high-performance of the conventional MC most of the time, while switching to the real-time scheduler only when timing guarantees risk being violated, which rarely occurs. In addition, unlike most existing real-time MCs, DuoMC enables the utilization of both private and shared DRAM banks among cores to facilitate communication among tasks. We evaluate DuoMC using a cycle-accurate multi-core simulator. Results show that DuoMC can provide better or comparable latency guarantees than state-of-the-art real-time MCs with limited performance loss (only 8% in the worst scenario) compared to the COTS MC.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**.

KEYWORDS

Memory Controller, DRAM, Predictability, Real-time Systems.

ACM Reference Format:
Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. 2021. DuoMC: Tight DRAM Latency Bounds with Shared Banks and Near-COTS Performance. In *The International Symposium on Memory Systems (MEMSYS 2021), September 27–30, 2021, Washington DC, DC, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3488423.3488431

1 INTRODUCTION

The exigent requirements of modern real-time embedded systems in terms of performance, power, and area are driving the adoption of multi-core platforms [2]. However, multi-core platforms come with their own challenges. One of the main hurdles is the interference due to the shared hardware resources among cores, including shared buses, I/Os, caches, and main memories. Predictably managing these resources to bound the resulting interference upon accessing them is not an easy task. Existing Commercial-Off-The-Shelf (COTS) arbiters managing access to these resources are designed with several complex optimizations aiming to achieve high average performance. Unfortunately, these optimizations result in extremely high latency in the worst-case, and thus, damage predictability. Performance predictability is important since it enables system designers to accurately reason about the behaviour of the system for crucial applications such as self-driving cars, automotive, and avionics.

This is because these COTS optimizations typically induce pathological scenarios that lead to extremely high latency in the worst case [20, 53], and thus, must be disabled to provide tight latency bounds. To address this challenge, several recent proposals introduced solutions redesigning these arbiters [6, 19, 22, 26, 39, 51] and controllers [1, 7, 8, 13, 34, 40, 43] to honor predictability by design. In contrast to COTS arbiters, these proposals provide strict timing bounds for Worst-Case Latency (WCL) incurred when accessing the shared resource by disabling most of the aforementioned performance optimizations. However, this *deteriorates* system performance. For instance, in Dynamic Random Access Memory (DRAM), we observe that most predictable memory controllers disable several optimizations usually deployed in COTS controllers such as reordering reads over writes and prioritizing requests that have DRAM page hits over page misses [20, 21, 27, 53]. Additionally, most of these predictable controllers disable data sharing through partitioning DRAM banks among cores to avoid intra-bank interference [13, 40]. This impairs the applicability of these proposals to modern embedded applications with increasing performance and data sharing demands [16].

To solve this fundamental conflict between average performance and predictability requirements, the authors of [35] recently introduced the Duetto reference model to manage shared resources in high performance multi-core real-time systems. Duetto is based on the observation that most of the time, requests to a shared resource through a COTS arbiter encounter latencies that are much smaller than the worst scenario, while the pathological scenarios that cause significant delays rarely occur in practice. Accordingly, Duetto pairs a real-time scheduler (RTSch) with a high-performance scheduler (HPSch) such that most of the time, the system will be enjoying the high-performance benefits of the HPSch, and only switch to the RTSch to ensure that all timing guarantees are honored if the resource becomes overloaded.

In this paper and inspired by the Duetto methodology, we propose DuoMC: a Memory Controller (MC) to manage accesses to DRAM in multi-core real-time systems. DRAM main memory is one of the most complex shared resources in multi-core architectures [10, 23, 34, 50] and it is one of the critical bottlenecks both from a latency as well as performance [17, 31, 37] perspectives. More in details, DuoMC makes the following contributions.

(1) DuoMC adopts the Duetto reference model [35], such that it can be modularly integrated into existing COTS MCs with minimal hardware modifications in the HPSch and without requiring detailed information on the internal behavior of the already implemented HPSch in the COTS platform. Unlike the simplified abstract SRAM resource in [35], DRAM is significantly more complex where a request requires multiple commands to be serviced, the data transmission for one request can happen concurrently with commands of other requests, and there are several timing constraints that must be tracked by the controller [45]. As a result, DuoMC extends and generalizes the conceptual model introduced in [35] to be applicable to more realistic shared resources existing in modern COTS Systems-on-Chip (SoCs). This generalization is discussed in details in Section 3.
(2) We propose a novel real-time MC scheduler, RTSch (Section 4) in which real-time guarantees are achieved by monitoring the latencies incurred by DRAM requests in the system and switching from a COTS HPSch to RTSch only in the rare cases when these guarantees are at the risk of being violated.
(3) Unlike most of the existing predictable MCs [7, 8, 13, 34, 40], DuoMC allows for communication among running tasks by declaring certain banks as shared to all requestors. Which banks are shared or private is configurable since it depends on the running set of tasks (Section 5.2). This is one of the key contributions of this work since most industrial embedded domains such as automotive and avionics [15] require communication among different tasks/processing components through shared data [5].
(4) We conduct a detailed timing analysis of DuoMC, which provides guaranteed bounds on the WCL suffered by any request to the DRAM (Section 5).
(5) We provide a detailed evaluation of DuoMC by implementing it in MacSim [28], a multi-core full-system, cycle-accurate simulator. Our results show that DuoMC suffers only 8% performance degradation across EEMBC [41] and IsolBench [48] benchmarks compared to a high performance memory controller (Section 7), while providing comparable WCL to state-of-the-art predictable MCs.

2 BACKGROUND AND RELATED WORK

We begin by providing the necessary background on the operation of DRAM, the associated MC, discuss related work on predictable DRAM MCs, and review the Duetto [35] methodology.

DRAM commands. Access to the DRAM is generally a two-stage process, where in the first stage, the row address is provided to load the requested row. This stage is called row activation and requires an *ACT* command. The second stage provides the column address to conduct the requested read/write operation through a *CAS* (*RD*/*WR*) command. Each DRAM bank also has a row buffer that caches the most recently accessed row in that bank. This enables future accesses to the same row to read/write from the buffer directly without activating the row. Such accesses are referred to as open accesses (or row hits) and are only composed of a *CAS* command. On the other hand, if an access requests a row different than the one in the buffer (a row miss), it has first to close the open row by writing it back to the DRAM cells in an operation called precharging. This requires the issuance of a *PRE* command. Afterward, the requested row can be fetched using an *ACT* command and finally, the read/write operation can proceed by a *CAS* command.

Table 1: DDR3-1600K/DDR4-2400U timing constraints [45].

Constraints	1600K	2400U	Constraints	1600K	2400U
Inter-bank Constraints (cycle)			Intra-bank Constraints (cycle)		
t_{RRD}: *ACT* to *ACT*	5	l=6,s=4	t_{RL}: read *CAS* to data	9	18
t_{FAW}: 4 *ACT* window	24	26	t_{WL}: write *CAS* to data	8	12
t_{RTW}: read *CAS* to write *CAS*	7	12	t_{WR}: write data to *PRE*	12	12
t_{WTR}: write data to read *CAS*	6	3	t_{RCD}: *ACT* to *CAS*	9	18
t_{WtoR}: write *CAS* to read *CAS*	17	19	t_{RP}: *PRE* to *ACT*	9	18
t_{CCD}: *CAS* to *CAS*	4	l=6,s=4	t_{RTP}: *CAS* to *PRE*	6	9
t_{Bus}: data transfer length	4	4	t_{RC}: *ACT* to *ACT*	37	57
			t_{RAS}: *ACT* to *PRE*	28	39

Timing constraints. All these commands (*PRE*, *ACT*, *RD*, and *WR*) have associated timing constraints that are mandated by the DRAM JEDEC standard [45] to ensure correct operation. Table 1 lists the constraints most related to this paper along with their description (note that all times in this paper are measured as multiples of the MC clock period, whose frequency is half the data rate of the device, i.e., for a 2400 device, the MC runs at 1.2GHz). As the table shows, some of these constraints apply to commands to the same bank (*intra-bank*), while others dictate the timings among commands to different banks (*inter-bank*). We say that a command becomes *intra-ready* (*inter-ready*) when it satisfies its *intra-bank* (*inter-bank*) constraints; a command is *ready* if it is both inter- and intra-ready. DRAM cells should be refreshed periodically to prevent data leakage by issuing refresh (REF) commands. The effect of refresh delays is usually not accounted for at the request-level analysis since it can be added as an additional delay term to the execution time of a task using existing methods [3, 50]. Accordingly and similar to previous works [20, 27, 53], we do not account for the refresh delay at the **request level**.

2.1 Memory Controller

Accesses to the off-chip DRAM are managed through an on-chip memory controller. The MC buffers incoming requests from different requestors into queues. Those queues are usually either per-requestor or per-bank. In this paper, we consider a system with M requestors: $\{P_1, ..., P_M\}$ and N DRAM banks $b_1, ..., b_N$. Note that a requestor can be any master entity in the system, including a CPU core, DMA engine, GPU, etc. Requests of each requestor are indexed based on the time at which they arrive, where $r_{i,j}$ is a request from P_i that arrives at timestamp $t^a_{i,j}$. Afterward, the MC performs three main operations as follows.

1) Address mapping. The MC translates the request address into DRAM physical address (e.g., which bank, row, and column to access). DuoMC allows for both assigning certain bank(s) to be private to a certain requestor as well as declaring certain banks to be shared among all requestors. This can be done either by the mapping function itself in the controller through configurable registers or through the virtual memory support in the OS [11, 12, 52].

2) Command generation. The MC in this step translates the request into low-level DRAM commands (i.e., a combination of *ACT*, *PRE*, *CAS*) based on the request type (load/store) and the state of the targeted bank. The MC keeps track of the state of all banks, which determines whether a bank has an open row in the row buffer and what is the address of this row (if any).

3) Command arbitration. Generated commands are buffered into per-bank *command queues*, and the MC arbitrates them to determine which ready command can be granted access to the DRAM device at every cycle. To conform to the JEDEC standard timing constraints, the MC also has to maintain a set of counters to track when a specific command is both intra- and inter-ready to be issued to the DRAM device. To increase performance, COTS MCs usually prioritize commands of row hits over row misses since they incur lower access latency. This policy is known as First-Ready First-Come-First-Serve (FR-FCFS) arbitration [44].

2.2 Predictable Memory Controllers

We next summarize the state-of-the-art efforts toward providing predictable DRAM behavior since they are the most related to this work. One direction is to analyze the DRAM in COTS platforms to provide latency bounds [20, 27, 53]. This direction has the advantage of being applicable to existing COTS platforms without requiring any hardware changes, which facilitates its adoption. In addition, it enables real-time systems to maintain the high performance of these COTS controllers. On the other hand, if the COTS controllers deploy some optimizations (e.g., read/write reordering), these solutions have to account for the corresponding significant latencies resulting from these optimizations, which leads to pessimistic latency bounds. Even worse, latency bounds become unattainable if COTS controllers deploy specific optimizations such as unlimited FR-FCFS reordering [20]. A second direction proposes to entirely re-architect the MC to achieve predictability by design [1, 7–9, 13, 25, 30, 32, 34, 40, 43, 47]. Despite achieving tight bounds compared to the first direction, this is realized by disabling most of the optimizations found in COTS MCs. Consequently, solutions following this direction suffer from major performance overheads. A recent third direction investigates the replacement of the commodity Double Data Rate (DDR) DRAMs that have inherent big latency variations with different emerging memory types that are more predictable, such as the reduced latency DRAM (RLDRAM) [17]. Although such direction is achieving promising results, DDR DRAMs are still the de facto standard for main memory, and hence, achieving predictability in systems adopting them remains an urgent challenge.

Compared to all these solutions, DuoMC promotes a completely different direction through the dynamic switching between the HPSch and the proposed RTSch based on the system behavior. This enables DuoMC to achieve comparable latency bounds to predictable MCs with almost no compromise of the system performance compared to COTS ones. There is also interest in the general architecture community to provide performance predictability and Quality of Service (QoS) for DRAM [29, 36, 46]; however, DuoMC is different in the sense that it provides strict guarantees under all possible scenarios (not only soft guarantees).

2.3 Duetto Methodology

DuoMC is inspired by Duetto methodology [35] which addresses the average performance and predictability trade-off in shared resource management in multi-core systems. Hereafter, we briefly review the background of the Duetto which help us to detail the proposed DuoMC in Section 3. The key idea behind Duetto is that it augments the COTS High-Performance Arbiter (HPA) with a Real-time Arbiter (RTA) such that both arbiters operate in parallel. The RTA must be analyzable in the sense that it provides strict latency bounds on requests. The system most of the time utilizes the performance gains of the HPA and only use the RTA decisions if there is a risk to miss the latency guarantees. Duetto Requests arriving from the requestors (such as DMA/Core/GPU, etc) are stored in the request buffers and then they can be arbitrated to access a specific shared resource. Once arbitrated, such requests are deemed to be finished after the shared resource processing time and thus will be removed from the buffers.

In addition to the HPA and RTA, Duetto comprises two other modules: *DTraker* and *WCLator*. *DTracker* is responsible to track these associated deadlines of requests at any clock cycle. Duetto aims to provide worst-case guarantees on the latency of all requests to the shared resource and assumes that a requestor can have multiple outstanding requests, and requests do not need to be serviced in the same order they are issued; therefore, it is crucial to bound the "processing latency" of requests. Hence, it associates a relative deadline to each requestor in the system which represents the maximum permitted processing latency for requests generated by that requestor.

WCLator module works as the core of the methodology by estimating the WCL of the outstanding request received from the requestors at run-time assuming that the arbiter selects the HPA in the current clock cycle and switch to RTA in all future cycles. If for any requestor, the estimated WCL from WCLator is lower than or equal to its deadline, then the WCLator continues with the HPA. Otherwise, it selects the RTA. This decision is based on the intuition that if the estimated WCL is less than the remaining time to service from DTracker, the deadline can still be met by continuing with the HPA (remember that the same logic will repeat every clock cycle). The key observation of the Duetto is that using online information on the state of the system (resource/requestors) allows us to greatly reduce the pessimism inherent in the static WCL computation.

Notice that the simple resource considered in [35] (bankized SRAM) only requires a single command to satisfy/finish a request; however, other realistic resources with a more complicated state machine such as DDR DRAM require arbiters to issue multiple commands (plus transferring the data) to declare a request to be finished. Therefore, DuoMC needs to extend the Duetto reference model as we discuss in details in the next section.

3 DUOMC: THE PROPOSED SOLUTION

In this section, we introduce the high-level architecture of the proposed DuoMC while Section 6 discusses some of the lower-level details that are specific to implementation. We start by discussing how DuoMC contributes to bounding the Worst-Case Execution Time (WCET) of a real-time task executing on a requestor/core P_i in Section 3.1. Second, we explain how DuoMC applies the Duetto reference model discussed in Section 2.3. Note that [35] demonstrated the model based on a much simpler resource, namely a bankized, SRAM-like memory with fixed access time and separate read and write data buses. DRAM poses a more complicated shared resource with multiple commands to service requests. The requests could require multiple commands to the device, which can happen in parallel with the data transmission to/from the DRAM. In addition, there exist timing constraints between commands that must be satisfied by the controller for the correct behavior of the device. For this reason, in Section 3.2 we extend the Duetto model. Specifically, 1) we generalize the timing model by distinguishing between completion and finish time of a request, and 2) unlike Duetto in [35], which is assumed to have direct access to the resource state, DRAM interface does not provide a mean to read its internal state; hence, the MC itself has to track this state. We detail the design of our novel RTSch in the following Section 4.

3.1 Preliminary: Task WCET Estimation

In real-time systems, timing guarantees depend on the WCET of tasks (schedulable entities). Following related work [14, 19], the WCET of a task running on core P_i can be represented as: $e_i = c_i + \mathcal{D}_i$, where c_i is the WCET of the task in isolation, i.e., with no interference from other tasks, while $\mathcal{D}_i$ is the total cumulative worst-case delay suffered by the task's requests to the shared resource under interference. $\mathcal{D}_i$ is calculated as the worst-case number of requests multiplied by the worst-case processing latency of each request [21]. In a multitasking system, the same approach can be applied [21, 27]: real-time scheduling theory can be used to determine the set of tasks that execute in a given busy interval (a scheduling window); e_i and $\mathcal{D}_i$ are then taken as the cumulative WCET in isolation and worst-case resource delay over all tasks in the window.

Since an out-of-order core might issue multiple simultaneous requests, and those requests might not be serviced in order, we need to formally define the finish time and processing latency.

Definition 1 (Finish Time). *The finish time $t^f_{i,j}$ of $r_{i,j}$ is the clock cycle after which $r_{i,j}$ finishes its data transfer.*

Definition 2 (Processing Latency). *For any request $r_{i,j}$, let $prec_j$ be the index of the request $r_{i,prec_j}$ of P_i with latest finish time among those that arrived before $r_{i,j}$ (i.e., such that $prec_j < j$). Then the* processing latency *of $r_{i,j}$ is* $\max\left(0, t^f_{i,j} - \max(t^f_{i,prec_j}, t^a_{i,j})\right)$.

A clarifying example is provided in Figure 1 (the concepts of completion time and oldest request will be introduced in the next section), where $prec_{j+1} = j$ and $prec_{j+2} = prec_{j+3} = j + 1$. Note that since $r_{i,j+2}$ finishes before $r_{i,j+1}$, its processing latency is zero. The intuition behind computing the latency in this manner is that if there is chain among requests, the processor cannot be blocked from when the first request is issued to when the last request is completed. Since we are looking to compute the length of this chain, anything completes within the chain has zero processing latency. In essence, Definition 2 ensures that the total latency of a chain of requests is equal to the sum of the processing latencies of the requests in the chain. The approach in [14] then formally guarantees that, as long as c_i includes the effect of timing anomalies due to out-of-order execution, the computed value of $e_i = c_i + \mathcal{D}_i$ is indeed an upper bound to the WCET of the task.

Since the task can have different request types to the resource (e.g., reads vs. writes and misses vs. hits), a tighter bound on $\mathcal{D}_i$ can be obtained by employing different latency bounds for each type of request [20, 49]. Note that because we are interested in the worst-case latency for the task and miss requests have higher latency than hit requests, we have to classify as a miss every request that

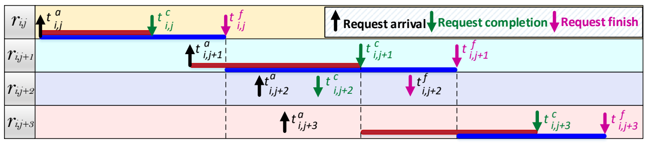


Figure 1: Processing latency shown in blue. Red bar represents the request being oldest. Assume that all previous requests $r_{i,k}$ with $k < j$ finish before $t^a_{i,j}$.

cannot be proven to access an open row. Specifically, we distinguish among four types of requests, where we use $\mathcal{T}(r_{i,j})$ to denote the type of a request $r_{i,j}$: 1) *RMP*: Read miss requests to a private bank. 2) *RHP*: Read hit requests to a private bank. Note that a private read request can be guaranteed to be a hit when analyzing a task only if under all possible program paths and initial hardware conditions, such a read will be issued after another read request to the same bank and row, and there is no possibility of closing the row before the request is serviced [4]. 3) *WMP*: Write requests to a private bank. Note that we are mainly interested in analyzing cores where memory requests are generated by last-level cache misses. Therefore, we consider the worst-case where writes are row misses: each write is generated by a cache replacement and write-back, and determining the precise order of replacements as to prove that the access is a hit is typically too difficult.4) *MSq*: Request to a shared bank, where q is the number of requestors that can access the bank targeted by the request. Since we cannot make any assumption on the interleaving of requests by different requestors, in general we cannot guarantee that such request is a hit; hence, we must consider it a miss. Finally, to obtain a static bound on the processing latency of each request type, we associate a (relative) deadline $D_i(\mathcal{T}(r_{i,j}))$ on the processing latency of requests of each type. Accordingly, we can bound the total delay $\mathcal{D}_i$ suffered by the task due to the shared resource by summing the deadlines of all requests generated by the task: $\mathcal{D}_i = \sum_{\forall j} D_i(\mathcal{T}(r_{i,j}))$.

3.2 DuoMC High-Level Operation

Figure 2 shows the conceptual architecture of DuoMC. As Section 2.1 explains, a COTS controller deploys a high-performance scheduler (denoted as HPSch in this paper) to arbitrate among different ready commands to the DRAM. Following the Duetto reference model, DuoMC requires minimal modifications and knowledge about this HPSch. On the other hand, DuoMC adds three main components to the system: as Figure 2 delineates, these are the RTSch, the *Deadline Tracker* (DTracker), and the DRAM *Worst-Case Latency Estimator* (WCLator), which are covered by blue pattern highlight in the figure. For concision, lower-level details that are specific to the implementation are discussed in Section 6. DuoMC operates through these components as follows:

1) ***Configurable deadlines:*** Each requestor has a configurable set of registers in DuoMC that are configured offline to the desired deadline for each of the four request types ($D_i(\mathcal{T}(r_{i,j}))$ from Section 3.1). Notice that if a requestor is not latency sensitive, the deadline can be relaxed. If further bandwidth regulation is required, there are other techniques such as MemGuard [54] that are orthogonal to DuoMC.

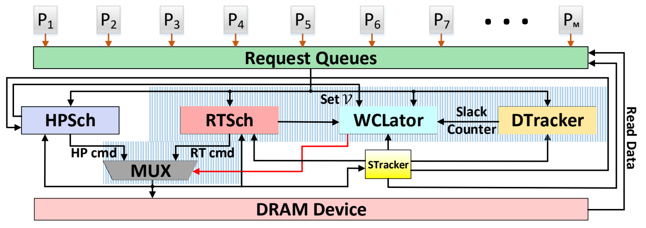


Figure 2: Conceptual Architecture of DuoMC including four main components.

2) ***Simultaneously running* HPSch *and* RTSch**: At run-time, DuoMC executes the two command schedulers in parallel: the HPSch and the RTSch. Every clock cycle, each scheduler outputs one DRAM command (possibly a no-operation *NOP*) based on its internal state, the queued requests, and the state of the DRAM. It is crucial to point out that having two controllers does not pose additional challenges from an electrical/implementation point of view, since both controllers share a single physical DRAM interface (PHY). COTS MCs need to keep track of the DRAM state to satisfy the correct operation according to the JEDEC standard. This includes the value of counters representing the remaining time until each timing constraint elapses, the command that is issued, and the states of banks (i.e., which row is open if any). This is maintained by the STracker in Figure 2.

3) ***Ensuring all deadlines are honored:*** In parallel, at each clock cycle the WCLator selects either the HPSch or the RTSch and issues the corresponding command to the DRAM device through a multiplexer. Therefore, switching between the two schedulers does not incur any overhead in terms of additional clock cycles. From an hardware overhead perspective, our design adds a MUX2:1 to the critical path, but this normally does not incur significant latency. The goal of the WCLator is to ensure that the processing latency of each request $r_{i,j}$ does not exceed its bound $D_i(\mathcal{T}(r_{i,j}))$; to this end, it receives as input a set of slack counters from the DTracker, which represent the remaining time until each request $r_{i,j}$ reaches its *absolute deadline* $d_{i,j} = \max(t^f_{i,prec_j}, t^a_{i,j}) + D_i(\mathcal{T}(r_{i,j}))$ (i.e., $d_{i,j}$ is the latest time by which $r_{i,j}$ must finish to satisfy its latency bound). We next detail the operation of the DTracker and WCLator, while the RTSch is covered in Section 4.

We begin by explaining how absolute deadlines are tracked. The ***DTracker*** only tracks the *oldest* request of each requestor P_i, according to Definitions 3 and 4. Intuitively, the completion time of a request marks the cycle at which the request has been fully processed by the controller by issuing its relevant commands; while the finish time of the request, as defined in Section 3.1, represents the time at which data is returned to the requestor (for a read). Hence, every request $r_{i,j}$ completes at the controller before it finishes at P_i; in details, following the timing constraints, we have $t^f_{i,j} = t^c_{i,j} + t_{RL} + t_{BUS} - 1$ for a read request, and $t^f_{i,j} = t^c_{i,j} + t_{WL} + t_{BUS} - 1$ for a write. Note that the order in which requests complete is the same as the order in which requests finish.

Definition 3 (Completion Time). *The completion time $t^c_{i,j}$ of $r_{i,j}$ is the clock cycle after the CAS for $r_{i,j}$ is issued.*

Definition 4 (Oldest Request of Requestor P_i). *We say that a request $r_{i,j}$ is* outstanding *in the interval $[t^a_{i,j}, t^c_{i,j})$ between its arrival and completion time. $r_{i,j}$ is* oldest *at time t if it is outstanding, and there is no other outstanding request $r_{i,k}$ of P_i with $k < j$ (i.e., with earlier arrival time).*

A request is put into the request queue once it arrives ($t^a_{i,j}$) and removed from the queue once it completes ($t^c_{i,j}$). Note that at any time t, there can be up to M oldest requests in the system, one per requestor. Also given that $r_{i,prec_j}$ is the request that finishes/completes last among those that arrive earlier than $r_{i,j}$, it follows that if $r_{i,j}$ finishes/completes after $r_{i,prec_j}$, then it must become oldest at time $\max(t^c_{i,prec_j}, t^a_{i,j})$, and remain oldest until it completes at $t^c_{i,j}$; otherwise, $r_{i,j}$ never becomes oldest and has zero processing latency. Intuitively, the DTracker only tracks the oldest request $r_{i,j}$ for each requestor P_i because any subsequent request cannot have an absolute deadline earlier than the completion time of $r_{i,j}$. Note that in the example in Figure 1, request $r_{i,j+2}$ with zero processing latency is never oldest. Also note that since there is no outstanding request of P_i in $[t^c_{i,j}, t^a_{i,j+1})$, no request of P_i is oldest in that interval.

The HPSch is designed to maximize average-case performance. In our implementation, the HPSch employs a FR-FCFS policy, as discussed in Section 2.1. On the other hand, the RTSch must be designed to provide tight latency bounds. The *Static Worst-Case Latency Bound* for each type of request represents the worst-case latency that a request can suffer assuming that the WCLator always selects the RTSch after the request becomes oldest.

Definition 5 (Static WCL Bound [35]). *For every requestor P_i and request type, we use $\Delta_i(\mathcal{T}(r_{i,j}))$ to denote an upper bound to the processing latency of $r_{i,j}$, assuming any possible state of the RTSch, request queues and timing constraint counters at time $\max(t^c_{i,prec_j}, t^a_{i,j})$, and that the RTSch is always selected from that cycle onward.*

Clearly, latency bounds can be met only if the bound in Definition 5 is less than or equal to the configured deadline that is used to calculate the task's WCET in Section 3.1, i.e., it must hold $D_i(\mathcal{T}(r_{i,j})) \geq \Delta_i(\mathcal{T}(r_{i,j}))$ for every requestor and request type. While the assignment $D_i(\mathcal{T}(r_{i,j})) = \Delta_i(\mathcal{T}(r_{i,j}))$ minimizes the execution time of a task, as we show in Section 7, using a looser deadline assignment can lead to improvements in performance. While optimizing the values of D_i is outside the scope of this paper, we specifically point out that there are requestors, such as DMA components and GPU, that are generally less latency-sensitive than general-purpose cores, and can thus tolerate higher latency bounds.

Finally, we discuss the ***WCLator***. At each clock cycle and for each requestor P_i, the WCLator computes a bound on the remaining processing latency of its oldest request $r_{i,j}$, assuming that in the current cycle the HPSch issues one command in the provided set $\mathcal{V}$ (discussed later in the section), while the RTSch is always selected afterwards. If for all requestors, the obtained bound is less than the slack counter for $r_{i,j}$ (meaning $r_{i,j}$ finishes by its deadline $d_{i,j}$), then the WCLator selects the HPSch; otherwise, it selects the RTSch. As formally proven in [35], this guarantees that every request meets its absolute deadline. Intuitively, this is because for a request to miss its deadline, it would have to be oldest, and thus tracked by

DuoMC; but then the WCLator would not select the HPSch if this could cause the request to miss the deadline.

Note that the WCLator logic is repeated every clock cycle meaning that as long as the conditions on the remaining processing latency of every oldest request are satisfied, the WCLator can keep selecting the HPSch. As we show in Section 5, **the key idea is that using on-line information on the state of the RTSch and outstanding requests allows the WCLator to compute an on-line latency bound that is much tighter than the static one**.

We discuss the command set $\mathcal{V}$ provided by the HPSch. Since both the HPSch and the WCLator operate in parallel, in general the WCLator cannot know the specific command selected by the HPSch in the current clock cycle. In this case, $\mathcal{V}$ can be simply constructed as the set of all commands that are legal, that is, do not violate any timing constraint and are used to satisfy an outstanding request. However, if the WCLator can access some information about the elected HPSch's commands, then $\mathcal{V}$ can be constrained, leading to better on-line estimation. Section 6 has an extended discussion about these adaptations.

A final note is related to the refresh operations. As discussed in Section 2, refresh delays are normally accounted for at the task level. Hence, our proposed design resets the DTracker slack counter to the deadline once a refresh operation finishes. This guarantees that the processing latency of any request unaffected by refresh (i.e., refresh has not happened in lifespan of the request) is bounded by the per-request deadline $D_i(\mathcal{T}(r_{i,j}))$ as discussed above; while any request affected by a refresh (i.e., refresh is happened in the lifespan of the request) suffers an additional latency equal to the refresh overhead plus the deadline.

4 REAL-TIME SCHEDULER (RTSCH)

To support the described DuoMC framework, we present a novel RTSch design. Compared to previous predictable MCs, we design the RTSch to provide tight bounds not only on the static WCL, but also on the on-line estimation for accesses to private banks and also shared banks. Specifically, we employ a dynamic command scheduler, where the three types of commands required to satisfy requests: *PRE*, *ACT* and *CAS*, are scheduled by three distinct command arbiters. Since our goal is to guarantee the latency of oldest requests, each command arbiter favors commands of oldest requests over non-oldest ones; however, to avoid limiting parallelism, the command arbiter can still issue a command of a non-oldest request if there is no oldest request with an intra-ready command of that type (commands that are not intra-ready cannot be issued and thus are not considered by the corresponding arbiter). The priority among requestors follows a predictable Round-Robin (RR) scheme in the MC, so that each oldest request of a requestor can be delayed by no more than one oldest request for each other requestor. To limit the delay incurred when switching the data bus direction, we employ a bundling scheme similar to the one first proposed in [7]; the *CAS* arbiter groups *RD* and *WR* commands, and issues them in rounds of the corresponding direction: read or write round (both referred as CAS round). Formally, the following rules capture the behavior of the RTSch in order to achieve the aforementioned goals (compared to the HPSch where there is no such rules; hence, no guarantees). Note that for simplicity, we present the rules and prove the latency bounds in Section 5 for a non-pipelined version of the controller. As we will discuss in Section 6, if the controller uses multiple pipeline stages, the computed latency must be amended to include an additional pipeline latency term equal to the number of stages - 1.

Rule 1. (Round Robin Arbitration.) The RTSch maintains a RR order of requestors. A requestor is removed from the RR queue after the oldest request of that requestor completes (i.e., after the *CAS* of its oldest request is issued), and it is inserted at the back of the queue either immediately, if it has at least one other outstanding request, or when its next request arrives. At any time t, we use hp_i to denote the set of requestors that have higher priority than P_i at t, that is, are ahead of P_i in the RR queue. Notice that the RR order among requestors is maintained entirely inside the MC.

Rule 2. (Bus Conflict Handling) When multiple commands of different types can be issued at the same time by the command arbiters, a bus conflict occurs among these commands and the priority is as follows: *CAS* > *ACT* > *PRE*. We pick this priority order since it matches the delay caused by each type of command (i.e., *CAS* commands cause the largest delay and are thus most critical).

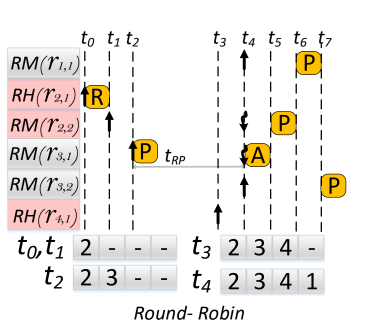
Rule 3. (Shared Bank Blocking.) All commands of oldest request $r_{i,j}$ of P_i targeting a shared bank b_r are blocked and cannot be issued if there exists a requestor $P_q \in hp_i$ whose oldest request targets b_r; the same applies if $r_{i,j}$ is non-oldest, except that in this case P_q can be any requestor (including P_i itself). This rule is required to ensure that the highest priority oldest request targeting b_r does not suffer intra-bank interference from other, lower priority requests targeting the same bank. In essence, given that no parallelism is possible among requests targeting the same bank, we force them to be serviced in strict RR order.

Rule 4. (PRE and ACT Arbiters Operation.) The *PRE* command arbiter arbitrates among non-blocked intra-ready *PRE* commands based on a two-level scheme: at the first level, it favors *PRE* commands of oldest requests over non-oldest requests. At second level, it employs the RR order of requestors. The *ACT* command arbiter uses the same logic applied to *ACT* commands.

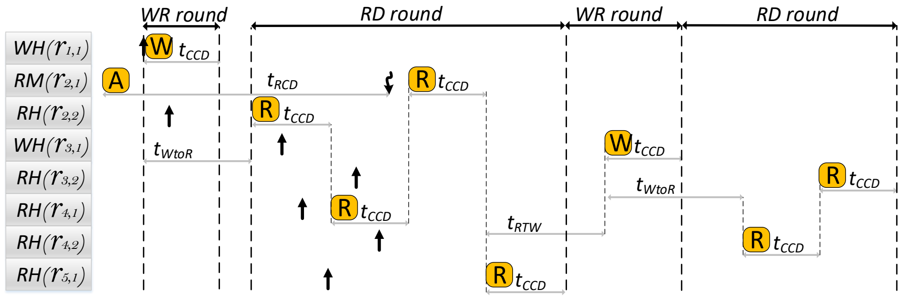
Rule 5. (CAS Self-Blocking.) To limit the length of each round, the *CAS* command arbiter keeps a service flag for each requestor. The service flag is set if a *CAS* of the oldest request of that requestor is sent, and reset when the round ends. If a service flag is set, *CAS* commands of requests of that requestor are considered blocked for the round. This ensures that no more than one oldest request per requestor can be issued in a round.

Rule 6. (CAS Round Starting and Ending.) A round ends t_{CCD} clock cycles after issuing a *CAS*, if there is no oldest request of the corresponding direction that is both intra-ready and unblocked. When a round ends, a new round starts immediately if there is any oldest intra-ready request; if any such request has the opposite direction of the old round, the new round has the opposite direction of the old one (this ensures that if there are both read and write oldest requests, their *CAS* commands are serviced in alternating rounds); otherwise, the new round has the same direction. If instead there is no oldest intra-ready request, then the next round starts either when an oldest request becomes intra-ready, or when the *CAS* of a non-oldest request is issued; the round direction equals the direction of the request.

Rule 7. (CAS Arbiter Operation Inside a Round.) Within a round, the *CAS* command arbiter arbitrates in RR order among non-blocked intra-ready *CAS* of the corresponding direction belonging



(a) Example describing Rules 1-4.



(b) Example describing Rules 1, 5-7.

Figure 3: Illustrative example: (a) *ACT*/*PRE* arbitration; (b) *CAS* arbitration. Curly down arrows represent the time commands of a miss request become intra-ready.

to oldest requests; unless there is no intra-ready *CAS* at all (either of the same or opposite direction) among oldest requests, in which case the arbiter selects in RR order among intra-ready *CAS* belonging to non-oldest requests. Note that based on Rules 6 and 7, if a *CAS* of a non-oldest request is sent, then the previous round must have ended so it must be the beginning of a new round.

Rule 8. (Always Starting with a Read Round.) Finally, note that if WCLator selects a command different than the one selected by the RTSch, it indicates that the WCLator is choosing HPSch in this cycle. In this case, we reset the state of the RTSch's *CAS* command arbiter to a read round with service flags cleared in order to favor open reads (for the possible switch to RTSch in the future): note that typically, the number of read requests generated by a task is significantly higher than the number of writes [18], and the write requests to DRAMs in modern architectures are due to last-level cache evictions, and hence, they do not stall the pipeline [20, 53]. The *PRE* and *ACT* arbiters are not affected since they do not have any state other than the RR order, which is not modified until the *CAS* of an oldest request is issued (request completes).

4.1 Illustrative Example for RTSch Rules

To illustrate the behavior of the RTSch, we next present two examples: Figure 3a focuses on the *PRE* and *ACT* arbiters according to Rules 1-4, while Figure 3b focuses on the *CAS* arbiter according to Rules 1, 5-7. The example in Figure 3a depicts 4 read miss requests: $r_{1,1}$,$r_{3,1}$,$r_{3,2}$ target private banks, $r_{2,2}$ targets shared bank b_s; and 2 read hit requests: $r_{2,1}$ and $r_{4,1}$ both targeting b_s. Requests targeting the shared banks b_s are highlighted in pink. We assume that there was no request before t_0. At t_0, $r_{2,1}$ arrives from requestor P_2. Hence, P_2 is pushed to the RR queue (**Rule 1**, requestor inserted). Since $r_{2,1}$ is a read hit and is intra-ready, its *CAS* is issued at t_0 and since it is oldest, P_2 is removed from the RR queue (**Rule 1**, requestor removed). At t_1, $r_{2,2}$ arrives from the same requestor P_2 and P_2 is again pushed to the RR queue. However, it is not intra-ready yet due to the *CAS* to *PRE* timing constraint; hence, bank b_s is busy and $r_{2,2}$ must wait until its *PRE* becomes intra-ready. Next, $r_{3,1}$ arrives at t_2, P_3 is pushed to the back of the RR queue and will remain there until its corresponding *CAS* is issued. The *PRE* of $r_{3,1}$ is issued at t_2. At t_3 an oldest, intra-ready read hit request from P_4 targeting bank b_s arrives and P_4 is pushed to the back of the RR queue. However, it will not be issued since there exists an oldest request that is not intra-ready ($r_{2,2}$) of a higher-priority requestor targeting b_s (**Rule 3**). At t_4 two requests arrive: $r_{1,1}$ which is an oldest read miss from P_1 (to its private bank) and $r_{3,2}$ which is a non-oldest read miss from P_3 (also to its private bank). P_3 already exists in the RR queue, while P_1 is pushed to the back of the queue at this point. At the same time, *PRE* of $r_{2,2}$ and *ACT* of $r_{3,1}$ become intra-ready. Although both requests are oldest and P_2 has higher priority than P_3, the *ACT* of $r_{3,1}$ is issued (**Rule 2**) first, while the *PRE* of $r_{2,2}$ is issued at t_5 since P_2 has higher priority compared to P_1 (**Rule 4**, second level). *PRE* of $r_{1,1}$ is issued at t_6 and *PRE* of $r_{3,2}$ is then issued at t_7 since $r_{1,1}$ is the oldest request of P_1 while $r_{3,2}$ is a non-oldest request (**Rule 4**, first level). Notice that $r_{4,1}$ will be serviced after higher priority $r_{2,2}$ is finished. Also note that we do not show the corresponding *CAS* commands of these requests, as we detail the *CAS* arbiter behavior in the next example.

Figure 3b shows the example of 8 requests to private banks: $r_{2,1}$ is a read miss, $r_{1,1}$ and $r_{3,1}$ are write hits, while the remaining requests are read hits. We assume that $r_{2,1}$ has already arrived, but its *RD* is not intra-ready yet. In the example, $r_{1,1}$ arrives first; a write round then starts and its *WR* is issued. The round ends t_{CCD} cycles after, since there are no intra-ready write oldest requests (**Rule 6**, round ends). Note that a new round does not start immediately, since there are no intra-ready oldest requests (note that $r_{2,2}$ is intra-ready, but it is not oldest). Once the t_{WtoR} data bus switching constraint elapses, *RD* requests become inter-ready; at this point, no oldest request is intra-ready yet, so the arbiter issues the *RD* of non-oldest request $r_{2,2}$ (**Rule 7**, non-oldest request) and a read round starts (**Rule 6**, round starts with non-oldest request). Note that because $r_{2,2}$ is non-oldest, the service flag for P_2 is not set, nor is P_2 removed from the RR queue (**Rules 1 and 5**). Afterwards, hit requests $r_{3,1}$, $r_{4,1}$, $r_{5,1}$, $r_{3,2}$ and $r_{4,2}$ arrive in this order, followed by the *RD* of $r_{2,1}$ becoming intra-ready; since the RR order depends on when requests become oldest, after $r_{5,1}$ arrives the order is $P_2 > P_3 > P_4 > P_5$. t_{CCD} cycles after issuing the *RD* of $r_{2,2}$, another *RD* can be issued; since $r_{2,1}$ is not intra-ready yet, $r_{4,1}$ is serviced instead, then $r_{2,1}$, and finally $r_{5,1}$ (**Rule 7**, RR order). Note that $r_{3,1}$ cannot be serviced in the read round since it is a write request, and $r_{3,2}$ cannot be serviced because it is not-oldest and there are intra-ready oldest requests (**Rule 7**, arbitration is between oldest requests of the corresponding direction). Once $r_{4,1}$ completes, P_4 is enqueued at the back of the RR queue, and its

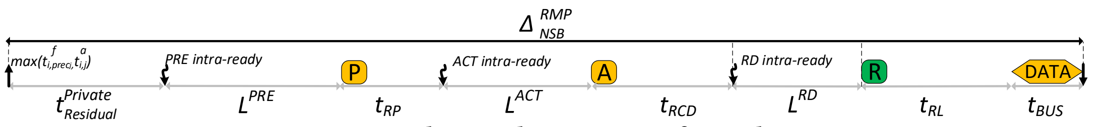


Figure 4: Request latency decomposition for read requests.

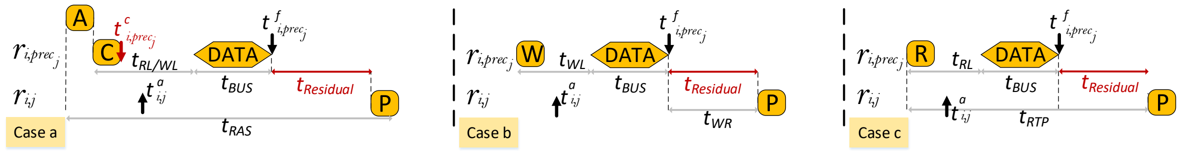


Figure 5: Case a, b, and c for the residual calculations for private bank access.

service flag is set (**Rule 5**); hence, $r_{4,2}$ is self-blocked and cannot be serviced in the first read round either, even if it is the oldest request of P_4 after $r_{4,1}$ completes (**Rule 7**, arbitration is between non-blocked requests). Once the read round ends, a write round starts since there is an intra-ready oldest write request $r_{3,1}$ (**Rule 6**, round starts with oldest request), which is then serviced. This causes P_3 to be enqueued at the back of the RR queue; hence, in the final read round, $r_{4,2}$ is serviced before $r_{3,2}$.

5 LATENCY ANALYSIS

In this section, we detail the latency analysis. We first derive the static WCL bounds for read requests to private banks in Section 5.1, and to shared banks in Section 5.2; we focus on discussing key novel results, while derivations for the very similar case of write requests to private banks are omitted due to space limitations. Then, we show how the static analysis can be modified to obtain the on-line WCLator estimation for the remaining processing latency in Section 5.3.

5.1 Static WCL Analysis: Private Banks

We begin by computing the static latency bounds $\Delta_i(RMP)$ for a read miss request $r_{i,j}$ of P_i targeting private bank b_r, which in the worst-case requires issuing a *PRE*, *ACT* and *CAS* commands; the read hit case $\Delta_i(RHP)$, comprising a *CAS* command only, is presented at the end of this subsection. We assume that $r_{i,j}$ finishes/completes after $r_{i,prec_j}$, otherwise its processing time would be zero; and based on Definition 5, we recall that the static bound is computed assuming that the RTSch is always selected starting at the time $\max(t^c_{i,prec_j}, t^a_{i,j})$ at which $r_{i,j}$ becomes oldest.

To derive $\Delta_i(RMP)$, we consider two cases, based on the status of the service flag for P_i when the *CAS* command of $r_{i,j}$ becomes intra-ready: the *self-blocking* case, which we denote with a *SB* subscript, corresponds to the service flag being set, while the *non-self-blocking* case (*NSB*) corresponds to the service flag being reset. We thus obtain:

$$\Delta_i(RMP) \quad = \quad \max(\Delta^{RMP}_{SB}, \Delta^{RMP}_{NSB}). \tag{1}$$

Note that because of Rule 1, we can remove the requestor index i from Δ^{RMP}_{SB} and Δ^{RMP}_{NSB} since our RTSch design employs a fair RR arbitration and the latency bound is the same for all requestors. Without Rule 1, every request would have different latency bounds.

We first analyze the more complex non-self-blocking case. We decompose the latency Δ^{RMP}_{NSB} into multiple terms, corresponding to its different commands and intra-bank constraints, as shown in Figure 4: (1) $t^{Private}_{Residual}$ is the worst-case latency from the start of the processing latency $\max(t^f_{i,prec_j}, t^a_{i,j})$ to *PRE* becoming intra-ready; (2) L^{PRE} is the maximum latency of *PRE* from the time it becomes intra-ready until it is issued; (3) t_{RP} is the *PRE*-to-*ACT* timing constraint; (4) L^{ACT} is the maximum latency of *ACT* from the time it becomes intra-ready until it is issued; (5) t_{RCD} is the *ACT*-to-*CAS* timing constraint; (6) L^{RD} is the maximum latency of *RD* from the time it becomes intra-ready until it is issued; (7) and finally $t_{RL} + t_{BUS}$ is the time required to complete sending the data. We next show how to bound the residual latencies, L^{PRE}, L^{ACT} and the *CAS* latency L^{RD}.

Residual Computation. The residual is computed based on the worst-case intra-bank constraints that can affect the *PRE* of $r_{i,j}$. Note that by definition of $r_{i,prec_j}$, once it completes at $t^c_{i,prec_j}$, either $r_{i,j}$ becomes oldest (if $t^a_{i,j} \leq t^c_{i,prec_j}$) or there must be no outstanding request of P_i. Since b_r is private, in the latter case, neither the RTSch nor the HPSch (given that it always issues legal commands) can issue any command to b_r between $t^c_{i,prec_j}$ and $t^a_{i,j}$; and starting at $\max(t^c_{i,prec_j}, t^a_{i,j})$ and until $r_{i,j}$ completes, the RTSch only issues commands of $r_{i,j}$ to b_r.

Therefore, it suffices to consider intra-bank timing constraints generated by the *CAS* of $r_{i,prec_j}$ itself, plus constraints generated by commands issued before such *CAS*. The detailed residual computation $t^{Private}_{Residual}$ is based on the three cases in Figure 5. In details, the three cases are: (a) If $r_{i,prec_j}$ does not target b_r, then in the worst case the HPSch could have issued an *ACT* command to b_r at time $t^c_{i,prec_j} - 2$. This triggers a t_{RAS} timing constraint; under the (worst-case) condition that $t^a_{i,j} \leq t^f_{i,prec_j}$, this results in a residual of $t_{RAS} - \min(t_{RL}, t_{WL}) - t_{BUS} - 1$. (b) If $r_{i,prec_j}$ targets b_r and is a write, then we need to consider the t_{WR} timing constraint between the end of data for a write *CAS* and *PRE* to same bank; again under the condition $t^a_{i,j} \leq t^f_{i,prec_j}$, this results in a residual of t_{WR}. (c) If $r_{i,prec_j}$ targets b_r and is a read, then we need to consider the t_{RTP} timing constraint between a read *CAS* and *PRE* to same bank; this results in a residual of $t_{RTP} - t_{RL} - t_{BUS}$. Taking the maximum of the three cases yields Equation 2.

$$\begin{aligned} t^{Private}_{Residual} = \quad & \max(t_{WR}, t_{RTP} - t_{RL} - t_{BUS}, t_{RAS} - \\ & \min(t_{RL}, t_{WL}) - t_{BUS} - 1). \end{aligned} \tag{2}$$

To facilitate the derivation of the on-line bounds in Section 5.3, we obtain L^{PRE} and L^{ACT} based on parameters k^{PRE}, k^{ACT} representing the state of the arbitration. Specifically, k^{PRE} is the number

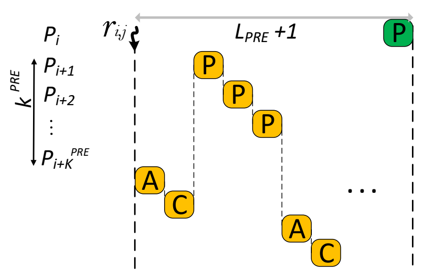


Figure 6: L_{PRE} **example.**

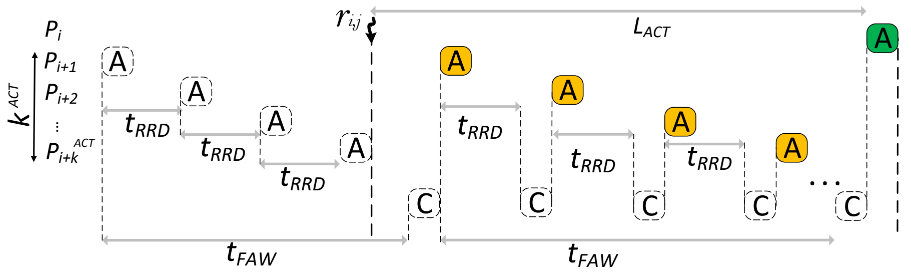


Figure 7: L_{ACT} **example considering** t_{RRD} **and** t_{FAW}**.**

of requestors in hp_i whose oldest request still requires issuing a *PRE*, while k^{ACT} is the number of such requestors whose oldest request still requires issuing an *ACT*. We make the following key observation:

Observation 6. *While $r_{i,j}$ is oldest, the values of k^{PRE} and k^{ACT} cannot increase as long as the RTSch is selected.*

Observation 6 holds because the RTSch always favors oldest requests. Hence, if the oldest request of a requestor in hp_i already issued a *PRE* (*ACT*), it will not require another *PRE* (respectively, *ACT*) until it completes; at which point the requestor will be enqueued at the back of the RR order and thus will have lower priority than P_i. Note that without Rule 4, this assumption was not true. Based on Observation 6, once the *PRE* of $r_{i,j}$ becomes intra-ready, the number of *PRE* commands that can be issued ahead of $r_{i,j}$ is bounded by k^{PRE}; the same holds for the number k^{ACT} of *ACT* commands that can be issued ahead of the *ACT* of $r_{i,j}$.

Computation of L^{PRE}. The worst-case latency pattern for *PRE* is depicted in Figure 6. Terms $\lceil \frac{L^{PRE}+1}{t_{RRD}} \rceil$ and $\lceil \frac{L^{PRE}+1}{t_{CCD}} \rceil$ in Equation 3 represent the command bus contention due to *ACT* and *CAS* commands, respectively, which are given higher priority compared to *PRE* command by the RTSch according to Rule 2. To bound such contention, we note that successive *ACT* commands are separated by at least t_{RRD} clock cycles, and successive *CAS* commands are separated by at least t_{CCD} cycles; while $L^{PRE}+1$ represents the maximum interference window where *PRE* commands (including the one of $r_{i,j}$) can be delayed by *ACT* and *CAS*. Adding the three terms yields the bound in the Equation 3.

$$L^{PRE}(k^{PRE}) = k^{PRE} + \left\lceil \frac{L^{PRE}(k^{PRE})+1}{t_{RRD}} \right\rceil + \left\lceil \frac{L^{PRE}(k^{PRE})+1}{t_{CCD}} \right\rceil. \tag{3}$$

Computation of L^{ACT}. The computation of L_{ACT} is more complex than L_{PRE}, since there exists *ACT*-to-*ACT* timing constraints t_{RRD} and t_{FAW}. The worst-case interference pattern is shown in Figure 7. Note that after the *ACT* of $r_{i,j}$ becomes intra-ready, only *ACT* of oldest requests accounted for in k^{ACT} can be issued; however, before the *ACT* becomes intra-ready, the RTSch could issue *ACT* of non-oldest requests. Specifically, in the worst case shown in the figure, four *ACT* commands of non-oldest requests are issued as late as possible before the *ACT* of $r_{i,j}$ becomes intra-ready, triggering an initial delay of $t_{FAW} - 3 \cdot t_{RRD} - 1$. Once the *ACT* of $r_{i,j}$ becomes intra-ready, no more than k^{ACT} other *ACT* commands can be issued before it; such commands cause a delay of either t_{RRD} each, or t_{FAW} every 4 commands. Finally, given that *CAS* commands are higher priority than *ACT*, but they cannot be issued in consecutive cycles, we incorporate the command bus contention by adding one unit of delay to each triggered timing constraint (including the initial delay). This yields the bound in Equation 4.

$$L^{ACT}(k^{ACT}) = t_{FAW} - 3 \cdot t_{RRD} + k^{ACT} \cdot (t_{RRD}+1) + \left\lfloor \frac{k^{ACT}}{4} \right\rfloor \cdot (t_{FAW} + 1 - 4 \cdot t_{RRD} - 4) \tag{4}$$

Note that in the worst case, the values of k^{PRE} and k^{ACT} are bounded by the total number of other requestors $M-1$; hence, when computing the static bound Δ^{RMP}_{NSB}, we must consider a latency $L^{PRE}(M-1)$ and $L^{ACT}(M-1)$.

Computation of *CAS* Latency L^{RD}. Let Round 1 to denote the round in which the *RD* of $r_{i,j}$ becomes intra-ready. We need to consider two possibilities, corresponding to Round 1 being a (1) read round, or a (2) write round; we use L^{RD}_{RD} to denote the latency bound in the first case, and L^{RD}_{WR} for the second case.

Case (1): since in the non-blocking-case the service flag for P_i is not set, it follows that the *RD* of $r_{i,j}$ must be issued in Round 1. As for the *PRE* and *ACT* computation, we use k^{RD} to denote the number of requestors in hp_i whose oldest request requires issuing a *RD*. We also use c^{inter}_{RD} to denote the inter-ready *RD* counter, that is, the number of cycles until a *RD* can be issued. Since the inter-bank constraint between *CAS* commands of the same direction is t_{CCD}, this yields the bound:

$$L^{RD}_{RD}(k^{RD}, c^{inter}_{RD}) = c^{inter}_{RD} + k^{RD} \cdot t_{CCD}. \tag{5}$$

Once again, for the static WCL computation we need to consider the worst case scenario and due to Rule 5, in each round maximum of M *CAS* commands can be issued (one from each requestor); hence, $k^{RD} = M-1$ which is the number of constraints between M *CAS* commands. There are two possible worst-case scenarios for c^{inter}_{RD}: (a) the *RD* command of an non-oldest request is issued immediately before the *RD* of $r_{i,j}$ becomes intra-ready, yielding $c^{inter}_{RD} = t_{CCD} - 1$; (b) the *RD* of $r_{i,j}$ becomes intra-ready as soon as possible at the beginning of the *RD* round, which is t_{CCD} cycles after the last *WR* is issued in a preceding write round; this yields $c^{inter}_{RD} = t_{WtoR} - t_{CCD}$. Hence, in the worst-case we have $c^{inter}_{RD} = \max(t_{WtoR} - t_{CCD}, t_{CCD} - 1)$.

Case (2): since Round 1 has write direction, the *RD* of $r_{i,j}$ will be issued in the following Round 2. In this case, once it becomes intra-ready, the *RD* of $r_{i,j}$ can only suffer interference from *CAS* commands of oldest requests: *WR* commands of oldest requests in Round 1, belonging to both higher and lower priority requestors, as

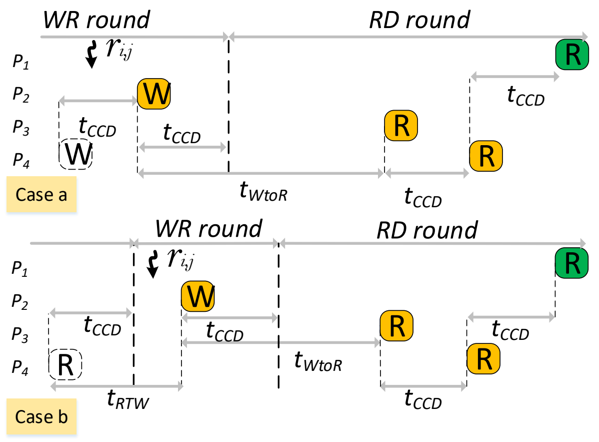


Figure 8: L^{RD}_{WR} example for $M = 4$ requestors, where $P_i = P_1$ and $k^{CAS} = 3$; in this example, two of the three interfering *CAS* commands are *RD* and one is *WR*.

well as *RD* commands of oldest requests in Round 2, belonging to requestors in hp_i; however, each requestor can only interfere once, since after sending the *CAS* of an oldest request, it is enqueued at the back of the RR queue according to Rule 1 and Rule 5. Therefore, let k^{CAS} to denote the number of interfering requests, where $k^{CAS} = M - 1$ in the worst case (again according to Rule 5). We again need to consider two possible worst-case scenarios, depicted in Figure 8: (a) the *WR* command of an non-oldest request is issued immediately before the *RD* of $r_{i,j}$ becomes intra-ready, resulting in the following bound:

$$k^{CAS} \cdot t_{CCD} + t_{WtoR} - 1. \tag{6}$$

(b) The *RD* of $r_{i,j}$ becomes intra-ready as soon as possible at the beginning of the *WR* round, which is $t_{CCD} + 1$ cycles after the last *RD* is issued in a preceding read round according to Rule 6 (note that if the *RD* became intra-ready one cycle before, the *RD* would be issued in the preceding read round); this yields:

$$t_{RTW} - t_{CCD} + (k^{CAS} - 1) \cdot t_{CCD} + t_{WtoR} - 1 = \\ (k^{CAS} - 2) \cdot t_{CCD} + t_{RTW} + t_{WtoR} - 1. \tag{7}$$

Taking the maximum of the two sub-cases we obtain:

$$L^{RD}_{WR}(k^{CAS}) = (k^{CAS} - 2) \cdot t_{CCD} + \\ \max(t_{RTW}, 2 \cdot t_{CCD}) + t_{WtoR} - 1. \tag{8}$$

Finally, we compare L^{RD}_{RD} and L^{RD}_{WR}. Note that by definition $k^{RD} \leq k^{CAS}$, and furthermore for all devices it holds $t_{WtoR} - 1 > \max(t_{WtoR} - t_{CCD}, t_{CCD} - 1)$. This yields $L^{RD}_{RD}(k^{RD}, c^{inter}_{RD}) < L^{RD}_{WR}(k^{CAS})$. Hence, unless we can guarantee that the *RD* of $r_{i,j}$ becomes intra-ready in a read round, we have to consider L^{RD}_{WR} in the worst case.

Non-self-blocking Latency. Combining all obtained bounds based on the latency decomposition in Figure 4 yields:

$$\Delta^{RMP}_{NSB} = t^{Private}_{Residual} + L^{PRE}(M - 1) + t_{RP} + L^{ACT}(M - 1) + \\ t_{RCD} + L^{RD}_{WR}(M - 1) + t_{RL} + t_{BUS}. \tag{9}$$

Self-blocking Latency. Finally, we consider the self-blocking case. Again, let the *RD* of $r_{i,j}$ become intra-ready in Round 1; since the service flag is reset whenever the round is switched or reset, it follows that Round 1 must be a read round, and that the *RD* of previous request $r_{i,prec_j}$ must have been issued in the round. Then, $r_{i,j}$ waits for the following write round (Round 2), and then its *RD* is issued in the next read round (Round 3) according to Rule 7. The corresponding worst-case scenario is depicted in Figure 9: the *RD* of $r_{i,prec_j}$ is issued at the beginning of Round 1; then $M - 1$ *RD* and *WR* commands of the other requestors are issued in Round 1 and 2; and finally by RR order, the *RD* of $r_{i,j}$ is issued first in Round 3 (notice that the *PRE* and *ACT* commands of $r_{i,j}$, which must be issued in Round 1, are not shown). Note that $r_{i,prec_j}$ finishes $t_{RL} + t_{BUS}$ after the beginning of Round 1; this is also the earliest time that the processing time of $r_{i,j}$ can start. Therefore, the latency of $r_{i,j}$ can be obtained by summing the length of the three rounds and subtracting $t_{RL} + t_{BUS}$, yielding:

$$\begin{aligned} \Delta^{RMP}_{SB} &= (M - 1) \cdot t_{CCD} + t_{RTW} + (M - 2) \cdot t_{CCD} + t_{WtoR} + \\ &\quad t_{RL} + t_{BUS} - (t_{RL} + t_{BUS}) \\ &= (2M - 3) \cdot t_{CCD} + t_{RTW} + t_{WtoR}. \end{aligned} \tag{10}$$

Note that the self-blocking case does not include any delay of *PRE* or *ACT*, but the number of *CAS*-to-*CAS* constraints t_{CCD} scales with twice the number of requestors M in the system. Since the *ACT*-to-*ACT* constraint t_{RRD} is never smaller than t_{CCD}, the non-self-blocking case leads to higher latency for miss requests.

Read Hit Latency. We again need to consider both the self-blocking and non-self-blocking case, yielding:

$$\Delta_i(RHP) = \max(\Delta^{RHP}_{SB}, \Delta^{RHP}_{NSB}). \tag{11}$$

Since the worst-case scenario for the self-blocking case in Figure 9 does not include the time for *PRE* and *ACT* commands, it also applies to a hit request, meaning that $\Delta^{RHP}_{SB} = \Delta^{RMP}_{SB}$. For the non-self-blocking case, we again decompose Δ^{RHP}_{NSB} into latency terms, similarly to Equation 9. Since $r_{i,j}$ is a hit, its *RD* command cannot suffer from intra-bank constraints, thus the residual is zero. This leaves the *RD* and data latencies only:

$$\Delta^{RHP}_{NSB} = L^{RD}_{WR}(M - 1) + t_{RL} + t_{BUS}. \tag{12}$$

Note that in this case, the self-blocking case has higher latency.

5.2 Static WCL Analysis: Shared Banks

We next compute the static WCL bound $\Delta_i(MSq)$ for a miss request $r_{i,j}$ of P_i targeting a bank b_r shared by q requestors. As in Section 5.1, we derive the bound based on a set of analysis parameters that capture the number of requests/commands that can interfere with $r_{i,j}$. Specifically, we use $\mathcal{S}$ to denote the set of requestors in hp_i whose oldest request also targets b_r, plus P_i itself; by assumption, for the number of requestors in $\mathcal{S}$ we have: $|\mathcal{S}| \leq q$. We also use $k^{PRE}_{\notin \mathcal{S}}, k^{ACT}_{\notin \mathcal{S}}, k^{RD}_{\notin \mathcal{S}}$ with the same meaning as k^{PRE}, k^{ACT}, k^{RD}, except that they only consider requestors that are not in $\mathcal{S}$. Note by definition we have: $k^{PRE}_{\notin \mathcal{S}} \leq M - |\mathcal{S}|$ (the same holds for $k^{ACT}_{\notin \mathcal{S}}, k^{RD}_{\notin \mathcal{S}}$).

We obtain $\Delta_i(MSq)$ by decomposing it into two latency terms: (1) the latency of the highest priority request in $\mathcal{S}$, served first by the RTSch, which we denote as Δ^{MSq}_{First}; (2) the latency of the remaining $|\mathcal{S}| - 1$ requests (including $r_{i,j}$ itself), which we denote as Δ^{MSq}_{Others}.

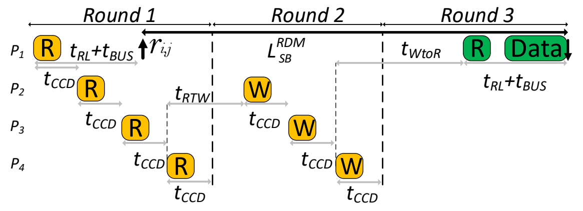


Figure 9: Worst-case scenario for a self-blocking read miss with $M = 4$.

To derive the latency terms, we make two key observations. First, the total number of *PRE* commands that can interfere with any of the requests in $\mathcal{S}$ is $k_{\notin\mathcal{S}}^{PRE}$; the same holds for *ACT* based on $k_{\notin\mathcal{S}}^{ACT}$. This is because by Rule 3, each requestor in $\mathcal{S}$ is blocked until every higher priority request in $\mathcal{S}$ completes; and once any requestor issues an interfering *PRE* or *ACT*, it cannot issue another one until its oldest request completes, at which point it ceases to be higher priority. Without Rule 3, the number of interfering requestors cannot be bounded. It remains to determine which request in $\mathcal{S}$ suffers interference. To this end, we use the following observation:

Observation 7. *For any non-negative values of k_1, k_2, it holds:*

$$L^{ACT}(k_1) + L^{ACT}(k_2) \le L^{ACT}(k_1 + k_2) + L^{ACT}(0) \quad (13)$$
$$L^{PRE}(k_1) + L^{PRE}(k_2) \le L^{PRE}(k_1 + k_2) + L^{PRE}(0) + 2 \quad (14)$$

Based on Observation 7, we can bound the *PRE* and *ACT* interference by simply assuming that the $k_{\notin\mathcal{S}}^{PRE}, k_{\notin\mathcal{S}}^{ACT}$ commands all interfere on the highest priority requestor in $\mathcal{S}$; except that we have to add 2 extra cycles to the *PRE* latency of each successive request in $\mathcal{S}$.

The second key observation is related to the value of k^{CAS} used in Equations 8. Once again, requestors in $\mathcal{S}$ are blocked until higher priority requests in $\mathcal{S}$ complete. Hence, when evaluating the *CAS* latency for the highest priority (first serviced) requestor in $\mathcal{S}$, we can use a value $k^{CAS} = M - |\mathcal{S}|$, rather than $k^{CAS} = M - 1$. However, as requests in $\mathcal{S}$ complete, the number of remaining requests targeting b_r decreases: for the second serviced request we need to consider a value $k^{CAS} = M - |\mathcal{S}| + 1$, and so on until $k^{CAS} = M - 1$ for $r_{i,j}$.

Based on both observations, and since the non-self-blocking case has higher latency for miss requests, we obtain:

$$\begin{aligned}\Delta_{First}^{MSq} \quad = \quad & t_{Residual}^{First} + L^{PRE}(k_{\notin\mathcal{S}}^{PRE}) + t_{RP} + L^{ACT}(k_{\notin\mathcal{S}}^{ACT}) + \\ & t_{RCD} + \max\big(L_{WR}^{RD}(M - |\mathcal{S}|) + t_{RL}, \\ & L_{RD}^{WR}(M - |\mathcal{S}|) + t_{WL}\big) + t_{BUS}, \qquad (15)\end{aligned}$$

$$\begin{aligned}\Delta_{Others}^{MSq} \quad = \quad & \sum_{l=1}^{|\mathcal{S}|-1} \Big(t_{Residual}^{Others} + L^{PRE}(0) + 2 + t_{RP} + L^{ACT}(0) + \\ & t_{RCD} + \max\big(L_{WR}^{RD}(M - |\mathcal{S}| + l) + t_{RL}, \\ & L_{RD}^{WR}(M - |\mathcal{S}| + l) + t_{WL}\big) + t_{BUS}\Big). \qquad (16)\end{aligned}$$

Note that L_{RD}^{WR} represents the worst-case bound for the *WR* where the write becomes intra-ready during a read round and can be computed similar to L_{WR}^{RD} except that read-related timing constraints are swapped for write-related timing constraints. Both equations use the same latency decomposition as for a miss request

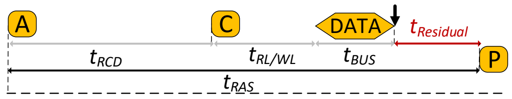


Figure 10: $t_{Residual}^{Others}$: computation of residual term for t_{RAS} constraint.

to a private bank (Equations 9), but we maximize over the read and write latencies for *CAS* and data since we do not know the direction of individual requests. It remains to determine the residual terms $t_{Residual}^{First}$ for the highest priority request, and $t_{Residual}^{Others}$ for the remaining $|\mathcal{S}| - 1$ ones. $t_{Residual}^{First}$ is the worst-case latency from the start of processing latency $\max(t_{i,prec_j}^{f}, t_{i,j}^{a})$ of $r_{i,j}$ to the *PRE* of the highest priority requestor in $\mathcal{S}$ becoming intra-ready and $t_{Residual}^{Others}$ is the worst-case latency between the finish time of a request in $\mathcal{S}$ and the *PRE* of the next request in $\mathcal{S}$. The same three intra-bank timing constraints (t_{RAS}, t_{WR}, t_{RTP}) used in the derivation of the private bank residual $t_{Residual}^{Private}$ must be considered. Given that b_r is shared, we cannot make any assumption on the commands that are issued to b_r before $r_{i,j}$ becomes oldest at $\max(t_{i,prec_j}^{c}, t_{i,j}^{a})$; while we know that only commands of requests in $\mathcal{S}$ can be issued afterwards. Hence, for the case of $t_{Residual}^{First}$, the worst-case scenario is that $t_{i,j}^{a} \ge t_{i,prec_j}^{f} > t_{i,prec_j}^{c}$ and either a *CAS* or *ACT* command is issued at cycle $t_{i,j}^{a} - 1$, resulting in Equation 17. On the other hand, in the case of $t_{Residual}^{Others}$, the timing constraint can only be generated by a command of the previous request in $\mathcal{S}$; hence, similarly to cases b and c in Figure 5, for the *WR* and *RTP* constraints we need to consider the time $t_{WL} + t_{BUS}$ or $t_{RL} + t_{BUS}$ required for the previous request to finish after issuing its *CAS*, resulting in the same residual terms t_{WR} and $t_{RTP} - t_{RL} - t_{BUS}$. Figure 10 shows the worst-case scenario for t_{RAS}, resulting in a term $t_{RAS} - t_{RCD} - \min(t_{RL}, t_{WL}) - t_{BUS}$. Taking the maximum of the three terms results in Equation 18.

$$t_{Residual}^{First} = \max(t_{WL} + t_{BUS} + t_{WR} - 1, t_{RTP} - 1, t_{RAS} - 1); \quad (17)$$

$$\begin{aligned}t_{Residual}^{Others} \quad = \quad & \max(t_{WR}, t_{RTP} - t_{RL} - t_{BUS}, t_{RAS} - t_{RCD} - \\ & \min(t_{RL}, t_{WL}) - t_{BUS}). \qquad (18)\end{aligned}$$

Finally, we have to determine the value of $|\mathcal{S}|$ under which $\Delta_i(MSq) = \Delta_{First}^{MSq} + \Delta_{others}^{MSq}$ is maximized. This is non-trivial, since increasing $|\mathcal{S}|$ adds more terms in Equation 16, but it decreases the maximum value of $k_{\notin\mathcal{S}}^{PRE}, k_{\notin\mathcal{S}}^{ACT}$ and the *CAS* latency. Hence, we simply numerically confirmed that for all devices, the bound is maximized by assuming the maximum number of contending requestors $|\mathcal{S}| = q$.

5.3 On-line WCLator Latency Estimation

We next discuss how the WCLator performs on-line estimation. Recall that the WCLator must compute the remaining processing latency (remaining time until the request finishes) for the oldest request $r_{i,j}$ of each requestor, assuming that the HPSch is selected at the current clock cycle t, while the RTSch is always selected afterwards. Since the round state, and hence the service flags, is reset whenever the WCLator issues a command that does not belong to the RTSch, on-line analysis only considers the non-self-blocking case.

We begin by considering a read miss request $r_{i,j}$ of P_i, targeting row rw in private bank b_r. We consider three cases, depending on which command $r_{i,j}$ needs to send next: *PRE*, *ACT* or *RD*. For each case, we use either c^{intra}_{PRE}, c^{intra}_{ACT}, or c^{intra}_{RD} to denote the time from t until the next command becomes intra-ready (or 0 if it is already intra-ready). We also consider the value of the inter-bank counter c^{inter}_{RD} at time t, as well as the values of k^{PRE}, k^{ACT} and k^{RD} at time t, as defined in Section 5.1. However, note that we do not use on-line information to bound the value of k^{CAS} (i.e., we consider the worst-case $k^{CAS} = M - 1$) because k^{CAS} includes requestors with both higher and lower priority, and we cannot predict when and which requests of another requestor might arrive in the future, causing it to be added to the RR queue with lowest priority. For each case, we list all possible commands that the HPSch can issue based on the state of bank b_r; for each command, we analyze the interference it causes on the next command of $r_{i,j}$, and derive a bound on the remaining latency of $r_{i,j}$ based on the latency components from Section 5.1. On-line, the WCLator computes all such bounds in parallel; it then discards the ones that do not apply based on the set $\mathcal{V}$ of possible commands of the HPSch; and finally takes the maximum of all remaining bounds to determine whether $r_{i,j}$ is guaranteed to complete by its deadline. Since the full command enumeration is rather pedantic and due to space limitations, here we only detail Case (1) (*PRE*) as an example. The remaining cases can be derived based on similar logic to the analysis for the other commands; we summarize it after covering Case (1). Similarly, we summarize how to derive the cases for read hits, write misses and shared bank requests after covering Case (1) for read misses.

Case (1): b_r is open on a row different than rw, then the RTSch needs to issue a *PRE* for $r_{i,j}$. The HPSch can issue either a *PRE* or a *CAS* for bank b_r (but not an *ACT*, since b_r is already open), but the *CAS* cannot target row rw, and thus cannot service $r_{i,j}$. It can also issue a *PRE*, *ACT* or *CAS* to some other bank, or a *NOP*.

- **(1.1)** *NOP*: **(1.1a)** if $c^{intra}_{PRE} > 0$, then a *NOP* cannot cause interference on the *PRE* of $r_{i,j}$, nor it changes the state of any bank; hence, the *PRE* delay for $r_{i,j}$ is bounded by $L^{PRE}(k^{PRE})$. Given that the *PRE* becomes intra-ready after c^{intra}_{PRE} cycles, and following the latency decomposition for a non-self-blocking request in Figure 4, we can then bound the remaining latency as:
$$L^{RD,1}_{online} = c^{intra}_{PRE} + L^{PRE}(k^{PRE}) + t_{RP} + L^{ACT}(k^{ACT}) + t_{RCD} + L^{RD}_{WR}(M-1) + t_{RL} + t_{BUS}. \quad (19)$$
Equation 19 represents the base latency computation for Case (1); all other sub-cases use either the same or a modified computation of $L^{RD,1}_{online}$. **(1.1b)** If $c^{intra}_{PRE} = 0$, issuing a *NOP* can waste a clock cycle and increase the *PRE* delay by 1; hence the latency is $L^{RD,1}_{online} + 1$.
- **(1.2)** *PRE* targeting bank $b_l \neq b_r$: the *PRE*-to-*PRE* interference is 1 cycle (command bus conflict only). Therefore, **(1.2a)** if $c^{intra}_{PRE} = 0$, we need to add 1 to Equation 19 to represent the extra cycle of delay. In addition, **(1.2b)** if at time t there exists a requestor $P_q \in hp_i$ whose oldest request targets b_l and requires a *PRE*, then we decrease the value of k^{PRE} by one, since the issued *PRE* satisfies such higher-priority, oldest request. However, **(1.2c)** if such oldest request requires only a *CAS*, then we need to increase the value of both k^{PRE} and k^{ACT} by one as the RTSch will need to now issue a *PRE*, *ACT* and *CAS* for such request.
- **(1.3)** *ACT* targeting bank $b_l \neq b_r$: since Equation 3 already accounts for command bus interference from *ACT* and *CAS*, the command cannot cause additional command bus interference. We thus compute the remaining latency bound according to Equation 19, except that we might need to change the value of either k^{PRE} or k^{ACT}, based on the following logic. **(1.3a)** If at time t there exists a requestor $P_q \in hp_i$ whose oldest request targets b_l, and the *ACT* issued by the HPSch opens a row different than the one targeted by such request, then it follows that the RTSch will need to again close b_l with a *PRE* to service P_q. Hence, the value of k^{PRE} must be increased by one. **(1.3b)** If instead the HPSch opens the row targeted by such request, then k^{ACT} must be decreased by one.
- **(1.4)** *CAS* targeting bank $b_l \neq b_r$: again there is no extra command bus interference, nor the bank state changes, so the latency is $L^{RD,1}_{online}$.
- **(1.5)** *PRE* targeting bank b_r: this is the *PRE* command required by $r_{i,j}$; then following the latency decomposition for $r_{i,j}$, the remaining latency is $t_{RP} + L^{ACT}(k^{ACT}) + t_{RCD} + L^{RD}_{WR}(M-1) + t_{RL} + t_{BUS}$.
- **(1.6)** *CAS* for bank b_r: if the *CAS* is a *RD*, then the latency is $L^{RD,1}_{online}$ as in Case (1.4). However, if it is a *WR*, we need to account for the t_{WR} intra-bank timing constraint between the data of such *WR* and the *PRE* of read request $r_{i,j}$. Hence, in this case the value of the c^{intra}_{PRE} counter in Equation 19 must be substituted with $\max(c^{intra}_{PRE}, t_{WL} + t_{BUS} + t_{WR})$.

The derivation for Case (2) and (3) (*ACT* and *CAS*) can similarly be carried out using base latencies $L^{RD,2}_{online}$ and $L^{RD,3}_{online}$, which are derived by summing the values of intra-counters c^{intra}_{ACT} or c^{intra}_{RD} with the remaining terms in the latency decomposition. For a hit request only Case (3) applies, and furthermore, it must hold $c^{intra}_{CAS} = 0$ since a hit request does not suffer intra-bank constraints. Therefore, the on-line estimation for a read hit can always use the better bound L^{RD}_{RD}, rather than considering L^{RD}_{WR} - this is indeed the reason why the RTSch resets the round state to read when the HPSch is selected. The on-line analysis for a write request to a private bank is similar; however, since the round is reset to read, we must always consider the worst-case bound L^{WR}_{RD} where the write becomes intra-ready during a read round. Finally, we discuss how to estimate the remaining latency of a request $r_{i,j}$ to a shared bank b_r. Following the discussion in Section 5.2, the on-line bound is still computed by summing the latency for the highest priority request in $\mathcal{S}$, and the latency for the other $|\mathcal{S}| - 1$ requests. For the latter, the WCLator uses the static bound Δ^{MSq}_{Others}; note that such bound depends only on the value of $|\mathcal{S}|$. For the former, instead of using Δ^{MSq}_{First}, the WCLator applies a logic similar to Cases (1)-(2)-(3) to the highest priority request in $\mathcal{S}$ to obtain a better on-line bound (with some parameter changes, e.g., $k^{PRE}_{\notin\mathcal{S}}$, $k^{ACT}_{\notin\mathcal{S}}$, $k^{RD}_{\notin\mathcal{S}}$ must be considered instead of k^{PRE}, k^{ACT}, k^{RD}, as discussed in Section 5.2).

Note that while the estimation includes several cases, in practice an efficient hardware implementation of the WCLator can be extremely fast. Notably, the WCLator can compute the bound for each case in parallel, compare each case against the slack counter for P_i, and then *and* the results together to determine if the HPSch can be selected. Furthermore, each case can be calculated quickly by pre-computing the various analysis terms and storing them in look-up table indexed based on the value of the various analysis parameters. We provide a more detailed implementation discussion in Section 6.

6 IMPLEMENTATION

To avoid slowing down the clock speed, it is imperative that DuoMC does not add significant extra logic to the critical path of the HPSch. Therefore, the WCLator cannot use the output of the last stage of the HPSch (the command that is arbitrated by HPSch); otherwise, the WCLator logic would need to be placed in **series** after the HPSch. For this reason, the set of commands $\mathcal{V}$ (shown in Figure 2 and used in Section 5) cannot be practically restricted to the unique command issued by the HPSch. Fortunately, WCLator can still operate in parallel with HPSch and still have some knowledge about $\mathcal{V}$ by leveraging the fact that most COTS MCs are pipelined. Since there are at least two sets of buffers (request buffers at an earlier stage and then command buffers), there are at least two pipeline stages to enable the buffering of requests, generating specific commands, and then buffering them into corresponding command buffers.

Typically, to optimize for performance, more stages are deployed. Early stages are used to generate one command either for each requestor, or each bank of the resource that can be operated in parallel; then, the last stage picks one of the generated commands. Hence, the set $\mathcal{V}$ can be restricted without requiring any modification to the HPSch. However, in this case, the analysis in Section 5 needs to add a fixed term to account for the extra delay suffered in the pipeline. In particular, each request suffers an additional *pipeline latency* equal to the number of stages - 1.

DuoMC supports different implementations based on the underlying COTS controller details. We consider three possible alternatives, based on the command set $\mathcal{V}$: (A) the most conservative design where WCLator make the decision based on the sets of commands that are legal in the current cycle without any further knowledge; (B) an improved design where $\mathcal{V}$ comprises one command per requestor per bank; (C) an ideal (but not practically implementable) design where $\mathcal{V}$ comprises a single command which is the one that HPSch selects. Our on-line latency analysis in Section 5.3 holds in all three cases, since it evaluates all commands in $\mathcal{V}$. However, for simplicity the WCL analysis assumes that a command of request $r_{i,j}$ can be issued immediately once the request arrives in the request queue at $t^a_{i,j}$.

Figure 11 shows results in terms of overall Instruction Per Cycle (IPC) for the three alternative designs listed above. We use a similar setup as in the evaluation Section 7: we consider $M = 8$ requestors contending for DDR3 1600K DRAM access; each requestor is assigned a private bank, and the foreground core executes the latency benchmark. For DuoMC, we first set all deadlines to the minimum possible value $D_i(\mathcal{T}(r_{i,j})) = \Delta_i(\mathcal{T}(r_{i,j}))$ from the static analysis, and then progressively increase them up to $4 \times \Delta$ (300% increase).

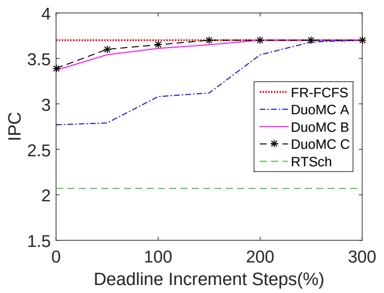


Figure 11: Overall IPC as a function of the relative deadline.

Note that here, design (B) behaves significantly better than (A). The reason is that looking at selected commands can allow us to exclude most of the worst analysis cases outlined in Section 5.3. On the other hand, (B) and (C) performs similarly, meaning that knowing the last stage of HPSch does not significantly improve the IPC compared to (B). Therefore, Section 7 follows (B).

We next discuss how the three modules added in DuoMC can be implemented. The DTracker module can be simply implemented by employing one counter per requestor, which counts down from the relative deadline $D_i(\mathcal{T}(r_{i,j}))$. For RTSch, in addition to utilizing the timing constraint counters, it needs to track the following information: 1) the RR order; 2) whether a requestor is blocked during the current round. All of these are simple to implement with a set of universal shift registers and flag registers [7]. For HPSch, note that once the WCLator selects the command from RTSch, the state of the HPSch can become invalid (for instance, if the RTSch closes a bank). Updating the internal state of the HPSch would require internal knowledge of its operation and possibly complex re-engineering. To avoid such complexity, we leverage the fact that a request remains in the request queue until it completes. Accordingly, once the WCLator selects a command from RTSch, we flush the command registers of the HPSch. In the next cycle, the HPSch will operate normally by reading requests from the request buffer and translate them into new commands taking into account the updated DRAM state. Such flushing does not usually require any modifications to the HPSch since most COTS controllers provide special operation registers (including flushing/reset capabilities) to recover from faults or unstable states [24, 42]. It also does not affect correctness, since the latency estimation depends only on the RTSch and the set of commands $\mathcal{V}$ that can be issued by the HPSch.

It remains to discuss the implementation of the WCLator. As shown in Section 5.3, the remaining latency estimation for each requestor P_i depends on various cases. However, the calculation in each of them comprises at most six terms: 1) the intra-ready counter for the next command of the request under analysis, which is known from the timing constraint counters; 2) L^{PRE} which depends on either k^{PRE} or $k^{PRE}_{\notin\mathcal{S}}$; 3) L^{ACT} which depends on either k^{ACT} or $k^{ACT}_{\notin\mathcal{S}}$; 4) the *CAS* latency, which depending on the case is either constant or depends on k^{RD} and c^{inter}_{RD} or $|\mathcal{S}|$; 5) Δ^{MSq}_{Others}, which depends on $|\mathcal{S}|$; 6) and a constant which can be computed off-line based on the value of timing constraints. To avoid computing each term on-line based on the corresponding equation, we pre-calculate it for every possible value of the parameters and store it in a look-up table. Note that the hardware required to calculate the estimation is fast and simple and can be implemented with a

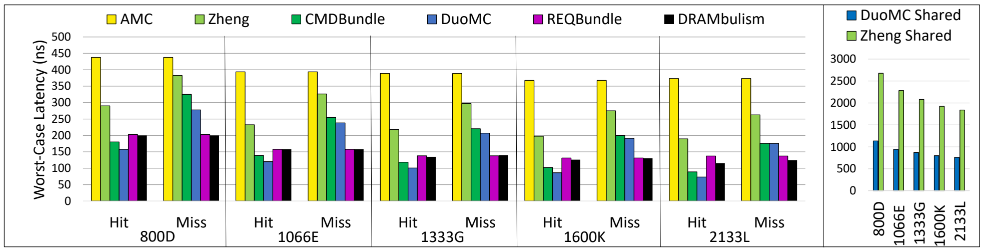


Figure 12: Analytical worst-case latency of read request (private and shared banks) across different speeds of DDR3 device.

small area footprint as it just needs to add six terms together, each of which can be stored in at most 11 bits. Instead of computing the maximum over all possible cases, the WCLator can then simply compare each case against the slack counter of P_i, which requires a 13-bit comparator in our implementation, and then *and* the results together to determine if the HPSch should be selected. The key advantage of this method is that each case can be evaluated in parallel, making the WCLator extremely fast. Finally, while the number of cases for each requestor P_i is significant, as noted in Section 5.3, only a subset needs to be considered in each clock cycle. Furthermore, several cases have the same or similar remaining latency, and their computation can thus be merged with simple combinational logic. Therefore, we believe that an optimized implementation can significantly reduce the number of latency terms that must be computed in hardware, albeit at the possible trade-off of adding extra combinational logic to the critical path.

7 EVALUATION RESULTS

Experimental Setup. We use MacSim [28], a multi-processor architectural simulator to model the requestors in our experiments. We incorporate superscalar x86 cores clocked at 1GHz. We run the experiments with multiple cache configurations; however, similar to [34], we show the results with bypassed caches in order to maximize the stress on the MC and DRAM device. For the memory subsystem, MacSim is integrated with the MCsim [33] memory controller simulator, employing a single-channel single-rank DRAM device. We use the virtual-to-physical address mapping capability in MacSim's front-end to implement bank partitioning/sharing among cores which can be achieved without running a complete OS manipulation. Since DuoMC is capable of handling shared banks, we also conduct experiments on DDR4-2400U to incorporate more banks in the experiments (DDR4 supports up to 16 banks compared to 8 banks in DDR3). The baseline HPSch deploys the FR-FCFS policy [38, 44], where requests to open rows are prioritized over requests targeting close rows in the same bank. This arbiter favors applications that can issue multiple concurrent memory requests. We assume that the HPSch provides to the WCLator a set $\mathcal{V}$ comprising one command per requestor per bank.

Considered MCs. We implement the proposed DuoMC inside MCsim. Since we are considering systems with strict guarantees, we compare against the following state-of-the-art real-time MCs, which also provide analytical WCL bounds and are implemented in MCsim: 1) Analyzable Memory Controller (AMC) [40], 2) CMDBundle [7], 3) DRAMbulism [34], 4) REQBundle [13], and 5) Zheng [50]. For ease of comparison, we do not model refresh operations.

Workloads. We use EEMBC benchmark suite [41] and also use two synthetic benchmarks: latency and bandwidth from IsolBench [48]. In all experiments, on the foreground core, we run one of the EEMBC or latency benchmarks and execute a bandwidth benchmark on the background cores to heavily stress the DRAM.

Analytical Worst-Case Memory Access Latency. Figure 12 represents the WCL bound on the read request latency of multiple state-of-the-art predictable MCs when there exist $M = 7$ requestors in the system and DDR3 devices with different speed bins are used (we employ DDR3 as some of the controllers we compare against cannot support DDR4). We use 7 requestors since DDR3 has 8 banks; in this way, we can use the extra bank as shared for MCs supporting shared banks, including Zheng [50] and RTSch. The remaining MCs only use 7 banks with one private bank per requestor. Note that we only show the latency bounds for read requests since they are larger compared to write requests. Based on the figure, we make three observations: (1) Zheng and AMC [40] perform worse than all other controllers; both are ones of the early designs which do not include recent optimizations to tighten the latency bound, such as bundling the commands/requests. (2) DuoMC and CMDBundle [7] have similar performance since both schedule individual commands and bundle *CAS* commands in specific rounds. The read hit bound for DuoMC is lower (15% for read hit and 7% for read miss in average) as it uses a more efficient round switching mechanism. (3) DRAMbulism [34] and REQBundle [13] also have similar performance since both controllers are optimized to execute requests in a pipeline, such that each miss request can only suffer interference on either *PRE*, *ACT* or *CAS* command, rather than all three of them. Consequently, both controllers perform better than DuoMC for miss requests, but worse on hit requests. Furthermore, note that the gap for miss requests tends to close for faster devices. Since DuoMC and Zheng support shared DRAM banks, we consider the eighth bank as shared among all the 7 requestors. As shown in the small figure, DuoMC performs better than Zheng because the latter considers each requestor with a request to the shared bank to be a virtual requestor such that a RR must be conducted among them in addition to the private banks' RR. This worsens the bound on the shared request by 141% for DDR3 1600K.

Measured Request Latency. Figure 13 delineates the latency suffered by read miss requests (in log_{10} scale) under FR-FCFS, RTSch and DuoMC. We consider $M = 8$ requestors contending for DDR3 1600K DRAM access; each requestor is assigned a private bank, and the foreground core executes the latency benchmark. For readability, Figure 13 only incorporates requests with latency longer than 80 cycles. The black dashed line represents the static WCL bound for DuoMC. For FR-FCFS controller, we observe noticeable latency

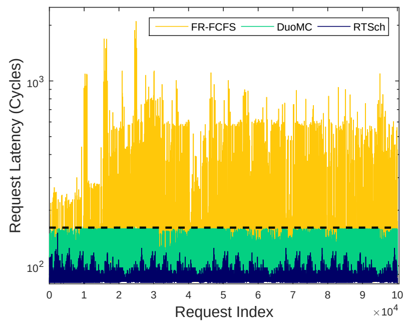


Figure 13: Request latency comparison amongst DuoMC, FR-FCFS, and RTSch. Only latencies greater than 80 cycles are shown. Note that Y axis represents the latency in log_{10} scale.

spikes all around the execution with a maximum latency of 2104 cycles. This is because FR-FCFS prioritizes requests that target an open row in DRAM, which can starve (theoretically) or delay for a significant amount of time (practically) requests that target close rows. On the flip side, RTSch guarantees the latency bound for all requests as expected. However, none of the requests come close to the static WCL bound since the analysis must make pessimistic assumptions on the state of the resource and RR order. Finally, we configure DuoMC with the minimum possible deadline $D_i(\mathcal{T}(r_{i,j})) = \Delta_i(\mathcal{T}(r_{i,j}))$ for each requestor. DuoMC stretches the latency of requests towards the relative deadline (dashed black line), because the WCLator estimates the latencies at run-time, and has more information on the system state. This allows DuoMC to select FR-FCFS as long as no request risks violating its deadline, thus significantly improving average performance compared to RTSch as we show next.

Average-Case Performance. Figure 14a shows the overall IPC of the system for each controller when the foreground core is running one of the EEMBC benchmarks, and a DDR3 1600K device is used. For DuoMC, we first set all deadlines to the minimum possible value $D_i(\mathcal{T}(r_{i,j})) = \Delta_i(\mathcal{T}(r_{i,j}))$, and then gradually increase them up to 2× of the minimum deadline (100% increase). By increasing the relative deadline in DuoMC, the framework will have more opportunity to select the HPSch. Under DuoMC with a minimum relative deadline (DuoMC-0), even though no request violates its deadline, the overall IPC of the system is very close to FR-FCFS controller. In particular, DuoMC-0 exhibits only 8% loss of performance over FR-FCFS on average while DuoMC-100 shows only 1% performance degradation. If D_i is large enough, the framework will almost always select the HPSch; hence, the performance of the system will be equivalent to the FR-FCFS controller. To compare, RTSch, which shows the best IPC over all other real-time MCs, causes 44% slowdown across the benchmarks compared to FR-FCFS. In order to allow the cores to communicate with each other, we use a DDR4-2400U device as it provides more banks compared to its DDR3 predecessors; specifically, it consists of 4 bank groups, each containing four banks resulting in 16 banks in total. We configure a system with $M = 7$ cores, each of them has exclusive access to 2 separate private banks, and two remaining banks are shared among all requestors. Since access time to the same bank group in DDR4 is longer than access to a different one, we assign the private banks for each requestor to reside in different bank groups to reduce access

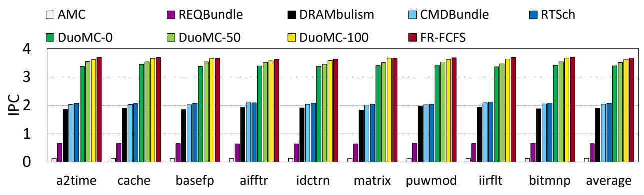


Figure 14: IPC for EEMBC benchmarks using DDR3-1600K.

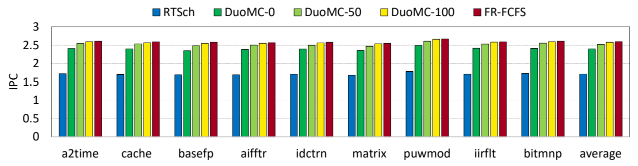


Figure 15: IPC for EEMBC benchmarks using DDR4-2400U.

time. In Figure 15, DuoMC shows an even better relative performance with DuoMC-0 compared to the Figure 14, resulting in only 7% slowdown compared to HPSch. For comparison, the figure also shows performance for the other controllers supporting DDR4.

We make the following observations: 1) the overall IPC of the system reduces compared to DDR3 and 8 requestors even though the number of banks is increased. This shows that DDR4 device does not perform better for the real world benchmarks since the additional bandwidth of DDR4 are not utilized as also discussed in [10]; 2) DDR4 devices are not suitable for memory sensitive applications as the access time is increased (11% increase in row hit and 14% increase in row miss compared to DDR3 devices) due to the bank groups in DDR4; 3) introducing shared banks reduces the parallelism in the DRAM device when requestors generate request to the shared banks (serviced sequentially). Therefore, the average performance of the system is degraded for both RTSch as well as HPSch; however, DuoMC-0 only shows 7% slowdown compared to HPSch.

8 CONCLUSIONS

We introduced DuoMC, a novel MC to manage DRAM memories in high-performance real-time embedded systems. DuoMC embodies two main contributions: 1) It generalizes the Duetto reference model [35] and extends it to COTS DRAM controllers as one of the most complex shared resources, where it allows the system to leverage the high-performance of the COTS high-performance scheduler most of the time, and only selects proposed real-time scheduler when timing guarantees are at risk of being violated. As a result, DuoMC achieves worst-case latency bounds that are comparable or better to that of existing real-time controllers at a very close performance to that of the COTS controllers; 2) It provides a mechanism to support both private and shared bank(s) combinations to enable shared-data communication among real-time tasks in the system.

REFERENCES

[1] Benny Akesson and Kees Goossens. 2011. *Memory controllers for real-time embedded systems.* Springer.
[2] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert Ian Davis. 2020. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*. York.
[3] Balasubramanya Bhat and Frank Mueller. 2011. Making DRAM refresh predictable. *Real-Time Systems* 47, 5 (2011), 430–453.

[4] Roman Bourgade, Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2008. Accurate analysis of memory latencies for WCET estimation. In *16th International Conference on Real-Time and Network Systems (RTNS 2008)*.

[5] Micaiah Chisholm, Namhoon Kim, Bryan C Ward, Nathan Otterness, James H Anderson, and F Donelson Smith. 2016. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 57–68.

[6] B. Cilku, B. Frömel, and P. Puschner. 2014. A dual-layer bus arbiter for mixed-criticality systems with hypervisors. In *IEEE International Conference on Industrial Informatics (INDIN)*.

[7] Leonardo Ecco and Rolf Ernst. 2015. Improved DRAM Timing Bounds for Real-Time DRAM Controllers Read/Write Bundling. In *Real-Time Systems Symposium*.

[8] Leonardo Ecco, Adam Kostrzewa, and Rolf Ernst. 2016. Minimizing DRAM Rank Switching Overhead for Improved Timing Bounds and Performance. In *Euromicro Conference on Real-Time Systems (ECRTS)*.

[9] Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. 2014. 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*. IEEE, 1–10.

[10] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. 2019. Demystifying complex workload-DRAM interactions: An experimental study. *ACM on Measurement and Analysis of Computing Systems* (2019).

[11] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. 2015. A survey on cache management mechanisms for real-time embedded systems. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–36.

[12] Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. 2019. Designing mixed criticality applications on modern heterogeneous mpsoc platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[13] Danlu Guo and Rodolfo Pellizzoni. 2017. A requests bundling DRAM controller for mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*. IEEE, 247–258.

[14] Sebastian Hahn, Michael Jacobs, and Jan Reineke. 2016. Enabling compositionality for multicore timing analysis. In *International conference on real-time networks and systems (RTNS)*.

[15] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. 2017. Communication centric design in complex automotive embedded systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[16] Mohamed Hassan. 2017. Heterogeneous MPSoCs for Mixed-Criticality Systems: Challenges and Opportunities. *IEEE Design & Test* 35, 4 (2017), 47–55.

[17] Mohamed Hassan. 2018. On the Off-chip Memory Latency of Real-Time Systems: Is DDR DRAM Really the Best Option?. In *IEEE Real-Time Systems Symposium*.

[18] Mohamed Hassan. 2020. 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 fuer Informatik.

[19] Mohamed Hassan and Hiren Patel. 2016. Criticality-and requirement-aware bus arbitration for multi-core mixed criticality systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 1–11.

[20] Mohamed Hassan and Rodolfo Pellizzoni. 2018. Bounding DRAM interference in COTS heterogeneous MPSoCs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).

[21] Mohamed Hassan and Rodolfo Pellizzoni. 2020. Analysis of Memory-Contention in Heterogeneous COTS MPSoCs. In *Euromicro Conference on Real-Time Systems*.

[22] Farouk Hebbache, Mathieu Jan, Florian Brandner, and Laurent Pautet. 2018. Shedding the Shackles of Time-Division Multiplexing. In *IEEE Real-Time Systems Symposium (RTSS)*.

[23] Sven Heithecker and Rolf Ernst. 2005. Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements. In *Proceedings. 42nd Design Automation Conference, 2005*. IEEE, 575–578.

[24] Intel. 2017. External Memory Interface Handbook Volume 2: Design Guidelines.

[25] Javier Jalle, Eduardo Quinones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. 2014. A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In *2014 IEEE Real-Time Systems Symposium*.

[26] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2011. Bus-aware multicore WCET analysis through TDMA offset bounds. In *2011 23rd Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE.

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

[28] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. 2012. Macsim: A cpu-gpu heterogeneous simulation framework user guide. *Georgia Institute of Technology* (2012).

[29] Y Kim et al. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.

[30] Yogen Krishnapillai, Zheng Pei Wu, and Rodolfo Pellizzoni. 2014. A rank-switching, open-row DRAM controller for time-predictable systems. In *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 27–38.

[31] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. 2013. Tiered-latency DRAM: A low latency and low cost DRAM architecture. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 615–626.

[32] Yonghui Li, Benny Akesson, and Kees Goossens. 2014. Dynamic command scheduling for real-time memory controllers. In *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 3–14.

[33] Reza Mirosanlou, Danlu Guo, Mohamed Hassan, and Rodolfo Pellizzoni. 2020. Mcsim: An extensible dram memory controller simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 105–109.

[34] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. 2020. DRAMbulism: Balancing Performance and Predictability through Dynamic Pipelining. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[35] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. 2021. Duetto: Latency Guarantees at Minimal Performance Cost. IEEE Design, Automation and Test in Europe Conference (DATE).

[36] Onur Mutlu, Justin Meza, and Lavanya Subramanian. 2015. The main memory system: Challenges and opportunities. *Communications of the Korean Institute of Information Scientists and Engineers* 33, 2 (2015), 16–41.

[37] Onur Mutlu and Lavanya Subramanian. 2015. Research problems and opportunities in memory systems. *Supercomputing frontiers and innovations* 1, 3 (2015).

[38] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith. 2006. Fair queuing memory systems. In *Proceedings of the 39th Annual IEEE/ACM international Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 208–222.

[39] Marco Paolieri, Eduardo Quiñones, Francisco J Cazorla, Guillem Bernat, and Mateo Valero. 2009. Hardware support for WCET analysis of hard real-time multicore systems. *ACM SIGARCH Computer Architecture News* 37, 3 (2009).

[40] Marco Paolieri, Eduardo Quinones, Francisco J Cazorla, and Mateo Valero. 2009. An analyzable memory controller for hard real-time CMPs. *IEEE Embedded Systems Letters* 1, 4 (2009), 86–90.

[41] Jason Poovey. 2007. Characterization of the EEMBC benchmark suite. *North Carolina State University* (2007).

[42] Qualcomm. 2016. Qualcomm Snapdragon 600E Processor APQ8064E Recommended Memory Controller and Device Settings Application Note.

[43] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. 2011. 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* (Taipei, Taiwan) *(CODES+ISSS '11)*. ACM, New York, NY, USA, 99–108.

[44] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. 2000. Memory access scheduling. *ACMSIGARCH Computer Architecture News* (2000).

[45] DDR3 SDRAM Standard. 2007. JEDEC JESD79-3.

[46] L Subramanian et al. 2016. BLISS: Balancing Performance, Fairness and Complexity in Memory Access Schedyuling. *IEEE Transactions on Parallel and Distributed Systems* 27 (2016), 3071–3087.

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

[48] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. 2016. 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.

[49] Zheng Wu, Y Krish, and Rodolfo Pellizzoni. 2013. Worst-case analysis of DRAM latency in multi requestor systems. In *IEEE 34th Real-Time Systems Symposium*.

[50] Zheng Pei Wu, Rodolfo Pellizzoni, and Danlu Guo. 2016. A composable worst case latency analysis for multi-rank dram devices under open row policy. *Real-Time Systems* 52, 6 (2016), 761–807.

[51] Man-Ki Yoon, Jung-Eun Kim, and Lui Sha. 2011. Optimizing tunable WCET with shared resource allocation and arbitration in hard real-time multicore systems. In *2011 IEEE 32nd Real-Time Systems Symposium (RTSS)*. IEEE, 227–238.

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

[53] Heechul Yun, Rodolfo Pellizzoni, and Prathap Kumar Valsan. 2015. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 184–195.

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