Adaptive Dual-Mode Arbitration for High-Performance Real-Time Embedded Systems

by

Reza Mirosanlou

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Alessandro Biondi Assistant Professor, Scuola Superiore Sant'Anna
Supervisor:	Rodolfo Pellizzoni Associate Professor, University of Waterloo
Internal Member:	Hiren Patel Professor, University of Waterloo
Internal Member:	Catherine Gebotys Professor, University of Waterloo
Internal-External Member:	Kenneth Salem Professor, University of Waterloo

Author's Declaration

Statement of Contribution

In what follows is a list of publications which I have co-authored and used their content in this dissertation. The use of the content, from the listed publications, in this dissertation has been approved by all co-authors.

- **Mirosanlou, R.**, Hassan, M., Pellizzoni, R.. (2021). Duetto: Latency Guarantees at Minimal Performance Cost. in 24th IEEE/ACM Design Automation and Test in Europe **(DATE'21)**. Grenoble, France, 2021 [75].
- **Mirosanlou, R.**, Guo, D., Hassan, M., Pellizzoni, R.. MCsim: An Extensible DRAM Memory Controller Simulator. in IEEE Computer Architecture Letters (**CAL**), pp. 105–109 2020 [73].
- **Mirosanlou, R.**, Hassan, M., Pellizzoni, R.. DuoMC: Tight DRAM Latency Bounds with Shared Banks and Near-COTS Performance. in ACM International Symposium on Memory Systems **(MEMSYS'21)**. Washington D.C., USA, 2021 [76].
- **Mirosanlou, R.**, Hassan, M., Pellizzoni, R.. Near-COTS Performance Under Predictable Cache Coherence. **Under Review**.

Abstract

Multi-core platforms can deliver substantial computational power together with minimum costs, compact size, weight, and power usage. However, multi-core architectures are shaking the very foundation of modern real-time systems, i.e. deriving the Worst-Case Execution Time (WCET) of the tasks. Modern embedded systems such as those deployed in the automotive and avionic fields face two difficult-to-resolve conflicting requirements due to the interference problem on the shared hardware components amongst cores: delivering high average-case performance and providing tight WCET. This challenge exists in different shared hardware resources including on-chip shared cache, hardware prefetchers, buses, and memory controller. The problem is mainly because various cores in the system interfere with each other while competing to access the aforementioned hardware components. While dedicated real-time controllers provide timing guarantees, they do so at the cost of significantly degrading system performance. This dissertation overcomes this trade-off by introducing Duetto, a general hardware resource management paradigm that pairs a real-time arbiter with a high-performance arbiter and a latency estimator module. Based on the observation that the resource is rarely overloaded, Duetto executes the high-performance arbiter most of the time, switching to the real-time arbiter only in the rare cases when the latency estimator deems that timing guarantees risk being violated. In this thesis, the Duetto paradigm is realized for different shared hardware resources. In the first part, I demonstrate Duetto on the case study of a multi-bank on-chip memory and discuss the foundation of the methodology. The methodology is concerned about designing the real-time arbiter in such a way that it is compatible with Duetto, deriving latency analysis, and designing the latency estimator module. In the second part, this thesis addresses the trade-off between maintaining cache coherence in multi-core real-time systems and improving average-case performance by proposing a novel coherency arbiter infrastructure and employing it in the context of Duetto. This is achieved by precisely engineering the multi-core hardware architecture and its underlying interconnect infrastructure such that data sharing is feasible for real-time systems in a manner amenable for timing analysis. The proposed solution provides near-to Commercial-Off-The-Shelf (COTS) performance and does not impose any coherency protocol modifications. The third part of this dissertation proposes DuoMC by applying Duetto to off-chip Memory Controller (MC) which is crucial since Dynamic Random-Access Memory (DRAM) main memory is one of the most complex shared resources in multi-core architectures and it is one of the critical bottlenecks both from latency as well as performance perspectives. As part of the MC evaluation, we release MCsim, an open-source, cycle-accurate simulator for memory controllers.

Acknowledgements

It is my absolute pleasure to thank all the people that made this dissertation possible. First, I would like to thank my advisor Professor Rodolfo Pellizzoni. He has been patient, accessible, and willing to help me on various issues. I am extremely grateful to my thesis committee members: Professor Hiren Patel, Professor Catherine Gebotys, Professor Kenneth Salem, and Professor Alessandro Biondi, for their guidance and feedback in developing my research. Each of them, in a peculiar way, helped me in my endeavor, supported my work, and significantly contributed to making it valuable. I am also very thankful to my collaborators who contributed to the presented work and my research in many ways. I thank Professor Mohamed Hassan, Professor Renato Mancuso, Professor Giovani Gracioli, Professor Marco Caccamo, and Dr. Rohan Tabish and Michael Guo; working with them on scientific papers was one of the most memorable in my Ph.D. A big shout-out to Professor Andrew Morton, who went above and beyond in supporting me during past years in many ways, and it was my honor to meet and work with such an amazing soul.

To my friends that shared with me days and nights, weekdays and weekends during my time at Waterloo, thank you for your camaraderie and for making the past four and a half years so memorable.

Finally, of course, is my rock, my inspiration, my world, my reason: without my wife, Homa Aghilinasab, none of this would ever have been possible. It is extremely difficult to properly thank you. I am indebted to you for dedication and sacrifices. At the dark times, Homa was always there, and always supported me unconditionally. To have a woman of such infinite talent and dynamism put her own professional career on hold to support the dreams of the man she's married is a powerful thing, and I have striven every day to live up to that.

Above all, I thank my family for their love and support. Words cannot express how grateful and happy I am to have such wonderful and kind people in my life. To my lovely parents, Goli and Hossein, you are my idols and the role models of my life. A thank also goes to my brother, Hani, who has always inspired me to see beyond academic life.

Dedication

To my family, for their support
with love and gratitude.

Table of Contents

List of Figures

List of Tables

List of Acronyms

WCET	Worst-Case Execution Time
WCL	Worst-Case Latency
PEs	Processing Elements
QoS	Quality of Service
FBW	Fly-by-Wire
LLC	Last-Level Cache
HRT	Hard Real-Time
SRT	Soft Real-Time
MC	Memory Controller
MSHR	Miss Status Holding Registers
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
DDR	Double Data Rate
SoC	System-on-Chip
RR	Round-Robin
GRR	Global Round-Robin

TDM	Time Division Multiplex
FCFS	First-Come-First-Serve
FIFO	First-In-First-Out
LOC	Lines-of-Code
FR-FCFS	First-Ready First-Come-First-Serve
COTS	Commercial-Off-The-Shelf
WCLator	Worst-Case Latency Estimator
DSM	Distributed Shared Memory
IPC	Instructions Per Cycle
OoO	Out-of-Order
C2C	Cache-to-Cache

Chapter 1

Introduction

1.1 Motivation

Real-time embedded systems, in different areas such as robotics, automotive, avionics, medical devices, and industrial environments [72] are increasingly deployed to provide timing guarantees. In such context, the correctness of the system is both a function of value as well as the time at which the results are produced [70]. Hence, timing constraints such as deadline should be provided for a real-time task in order to detect unacceptable results or maintain the Quality of Service (QoS). Timing analysis helps us to measure the WCET of the task. This is critical for real-time systems as it can be used in schedulability analysis to guarantee that tasks satisfy their timing constraints. To estimate the WCET, static or measurement-based analysis can be used [113]. The criticality of adhering to these timing constraints is determined based on the type of application. Hard Real-Time (HRT) applications such as Fly-by-Wire (FBW) in aircraft must satisfy the deadline, as the failure to satisfy timing constraints can result in severe consequences.

In the last two decades, there has been a constant augmentation in the popularity of embedded multi-core platforms due to real-time embedded applications claiming more processing power. This signifies a shaking point in the analysis and implementation of real-time systems. Multi-core platforms present benefits in terms of system cost and performance. However, such platforms introduce new challenges regarding accessing shared resources when multiple cores demand them simultaneously and create interference. This interference is a hurdle for real-time systems since the behavior of one core affects the temporal operation of other cores, which complicates the timing analysis of the system and makes it difficult to precisely derive the WCET of tasks. For instance, in [110], it was demonstrated that for the non-blocking cache structure used in modern multi-core systems, letting unlimited requests to enter the Miss Status Holding Registers (MSHR) of the cache leads to contention in MSHR. The result is blocking further accesses in the system and, hence, the WCET of the task could increase up to 27×. Experiments in [89] show that memory interference can contribute up to 300% to the WCET of a task, and memory bus interference can solely increase the WCET up to 44% [87]. In [39], it was demonstrated that the load/store on Last-Level Cache (LLC) could trigger coherency messages and increase the latency of such requests significantly or even make them unbounded.

Because of the existing shared resources in the multi-core platforms, there has been a large body of research efforts in the community investigating how to perform WCET estimation in such systems [112, 12, 58, 119, 83, 88, 19, 82, 9]. Normally, to tackle this problem, previous works [24, 34, 45] are proposing to estimate the WCET by dividing the execution time of the task into what the task does on the Processing Elements (PEs) (computation) plus the access latency of the shared resources such as, shared bus, memory controller, etc. Therefore, to determine the WCET of the task, it is crucial to bound the Worst-Case Latency (WCL) of requests accessing a shared resource. Obviously, tighter WCL for the resource accesses results in lower WCET of tasks. For the concern of this dissertation, the task analysis and evaluation of the request numbers can be done through either static program analysis or measurement-based approaches. A significant amount of interest exists in analyzing the WCL bound of resource accesses [33, 22, 32, 24, 42, 112, 39, 77, 56, 118, 37, 45, 84, 116].

Existing COTS arbiters managing access to these resources are designed with several complex optimizations aiming to achieve high average performance such as out-of-order execution, complex cache architectures, or branch prediction mechanisms. Unfortunately, these optimizations result in extremely high latency spikes in the WCL of accesses. This is because these COTS optimizations typically induce pathological scenarios that lead to extremely high latency in the worst case [121, 44], and thus, must be disabled to provide tight latency bounds. To address this challenge, several recent proposals introduced solutions redesigning these arbiters [18, 40, 46, 58, 83, 119] and controllers [1, 22, 23, 33, 74, 96, 84, 94] to honor predictability by design. In contrast to COTS arbiters, the goal of the real-time arbiter proposals is to provide strict timing bounds for WCL incurred when accessing the shared resource by disabling most of the aforementioned performance optimizations. However, predictably managing these resources to bound the resulting interference upon accessing them is not an easy task. In general, real-time arbiters expose bad average performance yet with tight latency bounds while COTS-based arbiters provide improved performance with large WCL spikes.

In summary, there exists a complex trade-off between the average-case performance and providing predictability for the shared resources in multicore platforms. Our key research question is: can we engineer a system that provides predictability while having minimal impact on average-case performance? To address this problem, this dissertation proposes a general reference model together with an informal methodology that enables the designers to address the predictability-performance trade-off in the multi-core platforms which I elaborate on in Section 1.2. Broadly, I aim to adhere to the generic platform in terms of average-case performance while also maintaining timing constraints by proposing using two arbiters for the same shared resource and switching between them based on the status and load of the system. It is important to point out that we are addressing this problem by proposing the reference model through architectural simulations and leaving the problem of having a full hardware implementation to future work.

1.2 Methodology

We introduce Duetto reference model in which we define two modes of execution: real-time mode and high-performance mode. In real-time mode, Duetto tries to optimize WCL of accesses by employing an arbiter that provides a tight bound for each request. In contrast, in high-performance mode, the execution model attempts to maximize the average-case performance and resource utilization of the system by using conventional architectural techniques that increase the overall performance of the system. Preferably, I propose a dual-mode arbitration model where most of the time, while the system is not at risk to violate timing constraints, the Duetto employs a mode optimized for the average-case performance.

More in detail, Duetto assigns WCL bounds to the requests, if the system is at risk to miss these WCL bounds in high-performance mode, the system switches to real-time mode to guarantee meeting these bounds. As mentioned before, the WCET of each task depends on the WCL of its resource accesses. We assume that there is a deadline associated with each request. The request deadline represents the maximum time that can elapse between a request arriving at the arbiter, and being serviced. If a requestor sends multiple requests, a deadline counter starts when the latest request arrives at the head of the transaction queue. This assumption is crucial, since otherwise, we may include queuing delay from the requests

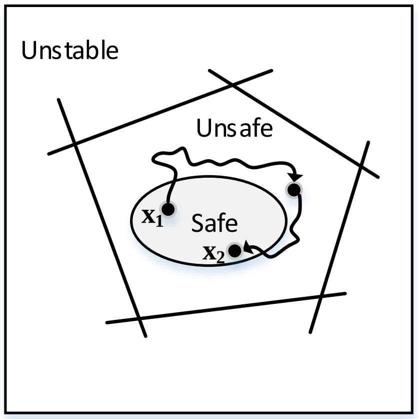


Figure 1.1: Simplex state space.

made by the same requestor. The deadline is configurable and we assume that the system designer first determines a request deadline for each particular request. Such an assignment should be performed based on the type of application (critical or non-critical tasks), and characteristics of the processing element (latency or bandwidth-bound). These deadlines might remain fixed or might be varied at run-time either on a partition or on a task basis by re-configuring the hardware. For instance, the designer might decide to perform a schedulability analysis on a core that runs a real-time task and based on the computation time and the number of requests in the task determines the maximum deadline value so the system remains schedulable. Note that, the approach that we consider in Duetto reference model to bound the WCET of the task is a request-driven approach. In details, the WCET of task is obtained by summing its execution time with the cumulative processing latency of memory requests. If the task performs at most M memory requests and its slack (how much its execution can be increased while still meeting timing constraints based on task-level schedulability analysis) is S, we can set the per-request deadline to S/M. The same idea has been presented in previous related work [40, 42].

Duetto has been inspired by other work in the area of safety for real-time systems, in particular, the simplex reference architecture [20] for control systems. The simplex architecture defines a methodology that allows the coexistence of an unverified, high-performance controller with a lower-performance but trusted safe controller. The simplex incorporates a checker component that monitors the stability of the system and switches the control output between the two controllers. Under normal operation, the system is far from the instability region, and it uses the output of the high-performance controller. However, if the checker component determines that an incorrect control command might cause the system to reach an unstable state at the next control step, it switches to the safe controller, which is guaranteed to maintain the system in the stable region. Once the system is again guaranteed to remain stable at the next step no matter the actuation command, the checker component switches back to the high-performance controller.

As Figure 1.1 shows, simplex defines three zones of operation for the system, and at any point in time, it can fall into one of these zones. The unsafe zone denotes when there is a *possibility* in which a wrong decision at the next step leads to violating the safety requirements. Likewise, with the right choice, it remains in an unsafe zone, or it can return to the safe zone. Note that the unsafe zone does not mean that the safety requirement is not met; instead, it simply means that there is a risk of going to an unstable zone. An unstable zone signifies that there is no guarantee for satisfying the safety requirement. The safe zone denotes the region in which regardless of any control action (whether switch or not), the system remains in the current zone or it moves to the unsafe zone as shown in Figure 1.1. It is important to note that while the system is running in the safe zone, there is no chance for the system to fall into an unstable zone. In the safe zone, we attempt to leverage high-performance mode since we will not become unstable in the next step. The real-time mode will be used in the unsafe zone since it can guarantee that there is no way the system goes to an unstable zone with any possible control action. Note that being in either the safe zone or unsafe zone means that the system still runs in stable conditions.

The important challenge here is that the semantics of Duetto and simplex are not equivalent. In detail, simplex is performed at the software level and it is capable of making the decision by looking at the system output/states on a large time scale. However, in Duetto, the instability of the system is defined based on a latency analysis. In other words, the state of the system will be unstable if it is not possible to guarantee its request deadline. Since the state in Duetto is too large, instead of defining safe/unsafe regions offline, Duetto has to operate on-line by applying the latency analysis to determine if the system goes to an unstable state. On top of that, the goal of Duetto is to control the output of high-speed hardware arbiters at the granularity of a clock cycle, requiring a high level of parallelism in the execution model that cannot be easily supported in the Simplex logic framework since the states in Duetto cannot be stored offline. Hence, the important challenge remains on how Duetto should operate at every clock cycle to guarantee that the latency of each request is not greater than the request deadlines. Most of the complexities that I will discuss in the coming chapters of this thesis are focused on keeping track of the resource's state and implementing the estimation such a way that it is fast enough and can be properly parallelized in hardware.

Since different shared hardware resources in a multicore system have their own specific constraints, the framework will have to be applied in different manners. In this thesis, I show that the proposed reference model could be successfully adopted on variety of shared resources, including shared buses (Chapter 3), on-chip memory (LLC) (Chapter 4), and off-chip memory (DRAM) (Chapter 5).

1.3 Structure of Thesis

Chapter 2 provides background on arbitration in multi-core platforms, main memory structure, and hardware cache coherence mechanisms. In addition, this chapter presents an overview of previous works related to each shared resource covered in the thesis. Chapters 3, 4, 5 are the main research contributions of this thesis. Chapter 3 introduces a novel hardware resource management paradigm called Duetto reference model. Based on the observation that the resource is rarely overloaded, Duetto executes the high-performance arbiter most of the time, switching to the real-time arbiter only in the rare cases when the latency estimator deems that timing guarantees risk being violated. In Chapter 3, I demonstrate Duetto on the case study of a simplified multi-bank memory. Chapter 4 proposes an arbiter for the shared memory in multi-core platforms aiming at providing predictable, coherent shared cache hierarchy solution, yet with a negligible performance, degradation compared to COTS solutions. Finally, Chapter 5 applies the Duetto reference model to one of the most complex shared resources, which is the main memory controller. On top of that, in Chapter 5, I introduce an off-chip memory device simulator, MCsim, an extensible and cycle-accurate MC simulator. Designed as an integrable environment, MCsim is able to run as a trace-based simulator as well as provide an interface to connect with external CPU and memory device simulators. I conclude this thesis in Chapter 6 that discusses future works based on the research contributions performed in the thesis and their limitations.

Chapter 2

Background and Related Work

This chapter presents the essential background required to continue with the other chapters of the thesis. In Section 2.1, we introduce the common arbitration schemes deployed in real-time systems and the related work. Next in Section 2.2 and Section 2.3, we cover the required background on the hardware coherency in shared caches along with the predictable coherency approaches. Finally, in Section 2.4, we study the organization of the memory controller as a shared resource among master entities, and for this reason, we provide the required background for DRAM main memory systems. In Section 2.5, we delve into the real-time memory controller design and discuss the most recent related work.

2.1 Background: Arbitration in Real-Time Systems

A contemporary multi-processor System-on-Chip (SoC) consists of a large number of components including streaming hardware accelerators and processors with caches that run several applications. As a result, shared resources, such as main memory and interconnect are introduced to reduce the cost. However, sharing the resources imposes interference among applications and makes it difficult to achieve satisfaction of the real-time requirements. Every shared resource can be used by any requestor in the system. Example of requestors includes CPUs and Direct Memory Access (DMA) engine. Arbiters are employed to decide which of the requestor should have the access (grant) to the resource at any point in time.

There has been a considerable number of arbitration mechanisms proposed by real-time research groups for shared resources. Traditional arbitration schemes such as Round-Robin (RR), fixed priority, and a hybrid of them are still used popularly. We review commonly used arbitration policies in traditional real-time systems to investigate their properties and performance. Particularly, we discuss fixed priority arbitration, simple First-Come-First-Serve (FCFS), RR arbitration including bare RR, Prioritized RR (PRR) [83], Weighted RR (WRR) [54], Harmonic RR (HRR) [119], and Harmonic Weighted RR (HWRR) [40] in addition to TDM-based arbiters [95, 28].

Fixed priority arbitration policy cannot provide a fair arbitration among the requestors since particular requestors are always prioritized over other masters. FCFS arbiters give the grants to the requestor that made a request earlier in time [122]. In other words, FCFS does not provide a fair arbitration among the requestors. First-Ready First-Come-First-Serve (FR-FCFS) is another conventional scheme that arbitrates between the entities who are ready and also at the same time arrived before every other requestor.

There have been several research efforts that investigate the criticality-aware arbitration on single-core platforms and multi-core platforms [42, 44]. Criticality is a designation of the level of assurance against failure needed for a system component. A mixed-criticality system has two or more distinct levels such as safety-critical, mission-critical, and low-critical [13]. There exist two approaches for incorporating the criticality in a system. [13] proposed comprehensive review on the mixed-criticality, which is mostly concerned about Vestal model [7]. The Vestal model assumes that there exist tasks or partitions with different criticality levels that are mapped to the same requestor and then tries to propose scheduling for them. On the other hand, the authors of [42] assigned the criticality requirements to the requestors instead of tasks.

Authors in [40] proposed a mixed critical platform for accessing the shared memory bus where they do not suspend low critical tasks when running in high mode. A software-based throttling is another scheme proposed in [86] to manage accesses to the shared main memory by assigning a budget to each core. Once the budget is exceeded for the non-critical cores, it throttles it to the guaranteed requirements of critical cores The technique in [111] arbitrates amongst memory requests from all tasks using conventional RR and FCFS policies.

Bare RR is dynamic and straightforward to implement. In addition, it is efficient and shares the resources equally among the requestors no matter what are their characteristics. The WCL of a request from any task can be bounded by the number of requestors in the system. However, in mixed critical systems where there are tasks with different criticality levels, RR cannot differentiate among the tasks. Researchers in [83] tried to cope with the deficiencies of RR by proposing prioritized RR. In such arbiter, RR is employed only for the critical tasks, and non-critical tasks can obtain access in slack slots. Slack slots are defined as the time that there is no request from the critical tasks in the system. The target of their solution is systems with the dual-criticality operation. Since they share these slots with non-critical tasks, there is no timing guarantee for non-critical tasks. [119] also proposed HRR to mitigate this problem by considering different slots to different tasks which maximize the system utilization.

Another research line concentrates on different arbitration policies for shared resources in order to eliminate interference. One example of this kind of arbitration policy is Time Division Multiplex (TDM), where the shared resource is assigned to a task or processing element for a predetermined time slot. In other words, it partitions access to the resource over time, and only a single requestor can acquire it at a time. When using TDM, the behavior of the requestors is entirely independent of the function of other requestors in the system.

WRR is able to allocate different amounts of service to processing elements based on their requirements [54]. Similar characteristics exist for TDM. The key difference between TDM and WRR is that TDM arbiters are non-work conserving. An assigned slot to a task will remain idle if there are no ready requests from this particular task even if there are ready requests from other tasks. Instead, WRR is work-conserving meaning that it assigns idle slots to the first task with a ready request.

Notice that the discussion in this section reviewed the arbitration mechanisms for a single resource such as a shared bus. However, in practice, there are sub-systems such as split-transaction bus or DRAM (covered in Chapters 4 and 5) which incorporate multiple parallel resources whose operation is interrelated. For such systems, a more complex arbitration scheme is required.

2.2 Background: Hardware Cache Coherence

Enabling data sharing is imperative in modern embedded systems. In these systems, massive amounts of data have to be collected (sensor fusion, cameras, etc), communicated through interconnect(s), and processed by various processing elements. As a result, recent efforts have been proposed to shift away from the independent task model, where tasks do not share data to a more-practical model that embraces data sharing and enables inter-core communication [17, 8, 63, 39, 105, 100, 101, 56].

Shared memory multi-core systems present a global address space in which processing cores can trade information and synchronize. When shared variables are cached in various caches concurrently, a memory store operation conducted by one processor can make data copies of the same variable in other caches invalid. The fundamental objective of a cache coherence protocol is to provide a coherent memory representation for the system so that each processor can observe the semantic impact of memory access operations performed by other processors in time. Various approaches can be employed in shared memory system implementation. Common hardware implementations extend traditional caching methods and use custom communication interfaces and specific hardware support.

2.2.1 Cache Controllers

Coherency in the system is maintained through coherence controllers, implemented as finite state machines placed inside each storage hierarchy, including private caches and LLC. Coherence amongst private caches is maintained through a cache controller. The cache controller is informed by any action from the core in the form of loads and stores and maintains the coherency by issuing coherence messages through the interconnection network. The receiving controller then performs corresponding coherence state transitions and issues a response back to the sender through the interconnection network. The types of requests and responses, as well as the state transitions, are dependent on the implementation specifics of each coherence protocol. Similarly, the LLC is also co-located with a state machine to implement coherence. The LLC controller typically does not issue coherence requests; instead, it receives and forwards requests or issues coherence responses through the interconnect.

Even though loads and stores instructions from the cores can read and write to specific bytes of data, the controllers operate in cache block granularity, meaning that coherency is maintained on a cache block instead of each byte. Hence, it is impossible to write to a cache block while another core processes a read/write request simultaneously. Therefore, updates to the values in a cache block must be propagated to all other cores in the system to preserve coherence. Efforts have been proposed for finer-grained hardware coherence [78, 99] or coarser-grained hardware coherence [14, 15].

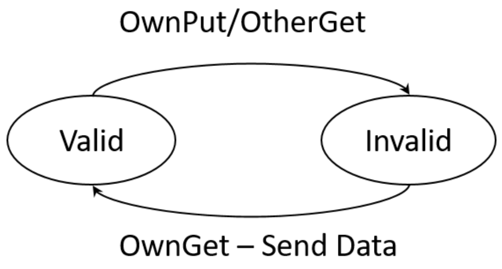


Figure 2.1: State transition diagram of a VI protocol.

2.2.2 Coherence Protocols

Coherence protocols determine the interactions between cache controllers and maintain all cache blocks up to date at any given time. A basic Valid-Invalid (VI) cache coherence protocol is shown in Figure 2.1. The VI protocol consists of two primary (stable) states, Valid (V), Invalid (I), and a transient state IV^D. Transient states are essential intermediate states between stable states to stall for either data or transaction acknowledgments to be received before entering the next stable state; however, they are not explicitly stated when describing a coherence protocol and for simplicity, we do not show the transient states in the figures. We assume the interconnect is a shared bus such that all coherence controllers broadcast (snoop) coherence requests.

At the beginning of execution, all cache blocks are in I state for all private controllers. Loads and stores from the core to the blocks in I state will cause the controller to issue a Get coherence request on the interconnect and transition to the transient state IV^D to stall for a response with data. Then, the shared memory controller observes the Get message and responds with data on the shared bus. Afterward, the requestor snooping on the bus copies the block to its own private cache and completes the load/store instruction. If other cores broadcast Get requests to the interconnect, the cache controller that has the block in V state will respond with its up-to-date data and invalidate itself transitioning back to I assuming that core-to-core transfer is possible; otherwise, any request should be directed to the shared memory. In VI protocol, only a single core is able to maintain the block in the V state at any given time. Any other loads/stores from other cores must first get the block in V state (up-to-date data) before they can perform their own requests. Despite VI protocol simplicity, the valid state only allows a single core to maintain a cache block with read permission at any point in time. This is true even if the block is unmodified by the core.

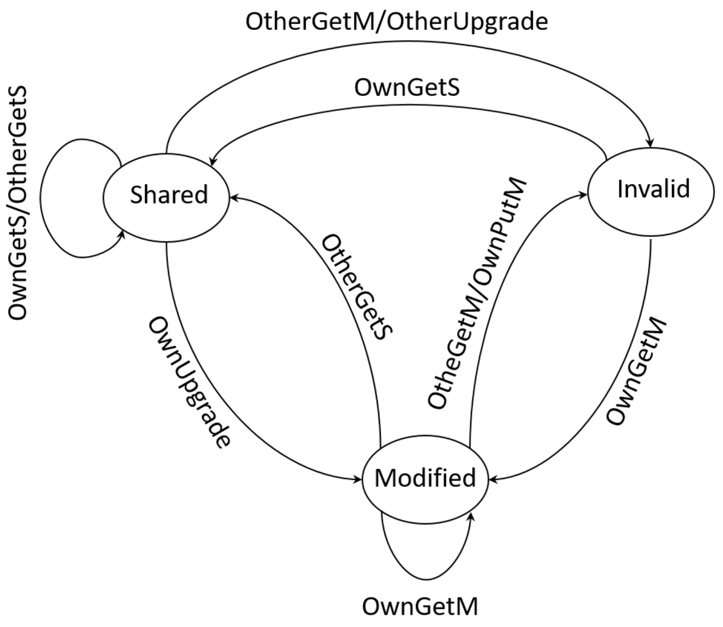


Figure 2.2: State transition diagram of an MSI protocol.

If multiple cores request to load one after another, each core sends Get requests on the interconnect and invalidate each other's cache blocks which incur unnecessary coherence transactions. In order to avoid this issue, numerous optimizations have been proposed. One example is dividing the V state into two distinctive states which are Modified (M) state and Shared (S) state to make MSI protocol. State M corresponds to blocks that have been modified (dirty) by a write request and state S corresponds to the blocks that are unmodified (clean) and is maintained by one or more cores. Figure 2.2 presents these states and the transitions amongst them which are implemented by the cache controller. Reading a cache line *A* requires the core to issue GetS(A) message on the interconnect and writing to *A* requires GetM(A). Upon replacement due to eviction or write-back to shared memory, a core needs to issue a PutM(A) message. If the core has cache line *A* in S state and tries to modify it, it issues an Upg(A) message. Every message on the bus will be observed by all cores' private caches as well as a shared cache controller. A core observing its own message is referred as Own, and any message initiated by other cores is referred to as Other. In Figure 2.2 a core that has a cache line in M state and observes other cores asking for the same line by OtherGetS message will move to S state since the cache line will be up to date in more than one cache controller. In spite of its simplicity, MSI is the foundation of most coherence protocols deployed in most existing COTS systems such as the MESIF protocol employed in Intel's i7 and the MOESI protocol employed in AMD's Opteron [48]. However, we detailed MSI coherence protocol in this section since MSI is covered in Chapter 4 and is used by related work on predictable coherence in real-time systems [49, 39],

2.2.3 Snoopy Protocols and Directory-based Protocols

There are two varieties of cache coherence protocols: snoopy protocols for bus-based systems and directory-based protocols for Distributed Shared Memory (DSM) systems. In bus-based multiprocessor systems, since all the processors can observe an ongoing bus transaction, appropriate coherence actions can be taken when an operation endangering coherence is detected. Protocols that fall into this category are called snoopy protocols since each cache snoops bus transactions to watch memory transactions of the other processors. Unlike snoopy protocols, directory-based protocols do not rely on the broadcast mechanism to invalidate/update stale copies. Instead, they maintain a directory entry for each memory block to register the cache sites in which the memory block is currently cached. The directory entry is usually maintained at the site in which the corresponding physical memory resides. Since the locations of shared copies are known, the protocol engine at each site can maintain coherence by operating point-to-point protocol messages. Thus, the removal of broadcast defeats a significant limitation on scaling cache coherent mechanisms to large-scale multiprocessor systems but incurs larger latency. For this reason, it makes little sense to employ a directory-based protocol for a small multicore. In this thesis, we focus on snooping bus cache coherence as it is commonly deployed in multi-core real-time systems with a small number of cores such as ARM chips [3].

2.3 Related Work: Predictable Cache Coherence

In general, predictability on the shared hardware resources in multicore real-time systems can be achieved by two approaches: 1) employing predictable bus arbitration (discussed in Section 2.1); 2) employing predictable cache coherence. The second category of works mainly focuses on enabling the predictable sharing of data among real-time core through hardware cache coherence [39, 30, 105, 57, 100, 56, 118, 6, 8, 37]. We find these approaches to be promising due to their performance benefits as well as transparency to the software stack. In addition, cache coherence is already the standard de facto in COTS multi-core platforms. These types of works normally require detailed formal modeling along with verification in order to verify their predictability. Moreover, these approaches are not extensible to other high-performance bus architectures such as those deployed in the COTS platforms including split-transaction interconnects. Using a split-transaction architecture allows the system to issue the requests (through coherence messages) and responses (i.e. data transfers) through different buses and manage them using different arbitration mechanisms.

Various approaches have been explored for shared cache to improve the predictability of the system such as preventing cores to access the data of the other cores in the system through isolation and partitioning [112], locking schemes [108], cache coloring [29, 31] or even splitting the data cache [98]. Notice that assuming private memory bank per core is not an optimized approach [39] as it suffers from multiple drawbacks: 1) They prevent sharing of data between tasks; hence, disabling any communication across applications or threads of parallel tasks running on various cores; 2) It might result in poor memory/cache utilization. For instance, it could increase the number of cache line eviction due to the partition space reaching its maximum capacity while other partitions might be underutilized; 3) It does not scale with increasing number of cores. While these approaches simplify the latency analysis of the task; they do not propose a solution to the problem of data correctness resulting from sharing memory space.

To exemplify, PMSI [39] was the first effort that introduced a set of novel transient coherency states to alleviate the unpredictable behavior of the conventional coherence protocol. However, it was achieved by significantly increasing the WCL bound of the requests. Recently, [56] proposed PMSI* and PMESI* which present a systematic approach towards designing predictable cache coherence mechanisms that achieve similar WCL bound compared to bypassing the shared cache and provide tighter WCL bound compared to original PMSI. They do so by capturing the key reasons behind the high WCL in existing predictable cache coherence and then apply micro-architectural extensions and protocol modifications in order to achieve tight WCL and high performance.

By decoupling the request and response bus and employing a split-transaction bus, [49] was able to achieve significantly tighter worst-case request latency bounds compared to PMSI [39]. PISCOT uses a predictable arbitration policy (TDM) on the request bus while employing conventional FCFS arbitration on the response bus. This caused PISCOT to exhibit a linear WCL similar to cache bypassing while improving average-case performance. Notice that our proposed approach in Chapter 4 (similar to PISCOT) does not require any

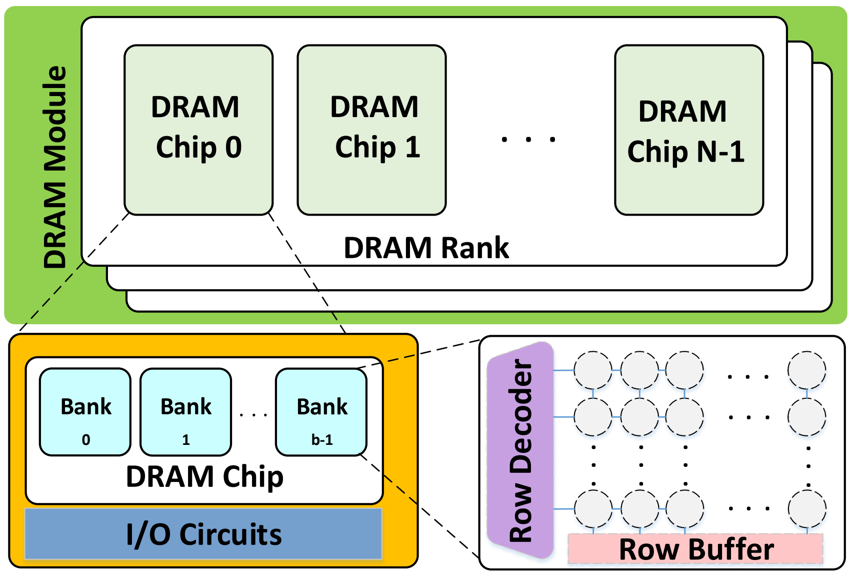


Figure 2.3: Internal organization of a DRAM module.

change or modifications on the underlying coherence protocol employed in the system since the predictability is achieved through re-designing the arbiter along with split-transaction interconnect.

2.4 Background: Main Memory

In this section, we describe the higher level of the DRAM design in terms of the hierarchy and modules, which leads to data transfers from/to the memory subsystem. These transfers impose latency on the PEs. To better understand the origin of memory latency, we first provide the fundamental background on DRAM organization and operations.

2.4.1 DRAM Organization

Figure 2.3 shows the internal organization of an off-chip DRAM. A DRAM device consists of one or more ranks, which share the address and data bus. Each *rank* contains multiple DRAM chips that share command signals among each other. DRAM chips are comprised of banks. A DRAM *bank* is a two-dimensional array of DRAM cells consisting of rows and columns. In addition, each bank contains sense amplifiers, which also work as a small cache, holding the most recently accessed row in that bank and are usually referred to as *row buffer*. Accesses to the off-chip DRAM are managed through an on-chip *memory controller*. A Double Data Rate (DDR) memory device is following a naming convention including generation, data rate, version. One example can be found in DDR3-2133L where three stands for the generation, 2133 accounts for data rate, and "L" represents the version of the device. The data rate is interpreted as mega-transfers per second. There are three basic DRAM commands. 1) **Activation** (*ACT*): upon accessing or receiving a request to a row that is not currently in the row buffer, the memory controller issues an ACT command to fetch the requested row from DRAM cells into the row buffer. 2) **Read-/Write** (*RD/WR*): once the requested row exists in the row buffer, the memory controller issues a *RD/WR* to conduct the read/write operation on the requested column(s) from the row buffer. This is usually referred to as a *CAS* command. 3) **Precharge** (*PRE*): If another row than the requested one resides in the row buffer, it has to be written back to the DRAM cells before activating the requested row. This operation is referred to as *precharging* and is conducted using a *PRE* command. In addition to these three operations, the memory controller has to periodically *refresh* DRAM cells to avoid data leakage using a *REF* command.

2.4.2 Memory Controller Operations

Accesses to the off-chip DRAM are managed through an on-chip memory controller. The MC buffers incoming requests from various processing elements in the system as well as other requestors such as DMA into the queues. Those queues are usually either per-requestor or per-bank. Note that a requestor can be any master entity in the system, including a CPU core, DMA engine, GPU, etc. 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). Memory controllers might allow 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 [29, 31, 120].

2. **Access Scheduling:** Once requests arrive at the controller, they are queued in a *request queue*, which might depend on the issuing requestor in addition to the target address. For instance, many memory controllers assume per-bank request queues [33, 22]. Memory controllers deploy arbitration both at the request level (using a request scheduler) and the command level (using a command scheduler). The *request scheduler* arbitrates among requests from different processing elements (requestors) to pick a request and translate it to its corresponding commands (*command generation*) and send it to the corresponding *command queue*. Finally, the *command scheduler* arbitrates among ready commands to elect a command to issue to the DRAM.

3. **Command Generation:** Command generator generates the command sequence for each request and places the generated commands in the corresponding ***command queue***. The generated commands depend on the deployed policy by the controller to manage the row-buffer, which is known as row-buffer policy or page policy. Two common page policies are open-page and close-page. Under the open-page policy, the memory controller leaves the data in the row buffer available for future accesses that might hit in the row buffer and hence encounter a lower access latency. On the other hand, the close-page policy writes back the data in the row buffer to the DRAM cells immediately after each access. Since consecutive memory accesses usually exhibit locality, the open-page policy generally improves average-case performance over close-page policy. However, if a request's row is different from the row in the row buffer, a *bank conflict* occurs and this request has to wait first for the *PRE* operation. Therefore, aside from the idle bank case (where the row buffer does not have any data), a request under the open-page policy has two scenarios. 1) A close request, whose row is different from the one in the row buffer; thus, the command generator has to generate three commands for this request: *PRE*, *ACT*, and *RD/WR*. 2) An open request, which hits in the row buffer, and thus, consists only of a *RD* (or *WR*) command.

2.4.3 DRAM Timings

All these commands (PRE, ACT, RD, and WR) have associated timing constraints that are mandated by the DRAM JEDEC standard [106] to ensure correct operation. Table 2.1 lists the constraints most related to this work along with their description (note that all times in this chapter are measured as multiples of the MC clock period, whose frequency is

Table 2.1: JEDEC DDR3/DDR4 timing constraints.

Constraints	1066E	1333G	1600H	1600K	1866K	2133L	2400U
Inter-bank Constraints (Cycle)							
t_{RRD}: Row to row activation delay	4	4	5	5	5	5	l=6,s=4
t_{FAW}: Activation window	20	20	24	24	26	27	26
t_{RTW}: Read to write switching time	6	7	7	7	8	8	12
t_{WTR}: Write data to read time	4	5	6	6	7	8	3
t_{WtoR}: Write to read switching time	14	16	17	17	20	22	19
t_{CCD}: Column to column time	4	4	4	4	4	4	l=6,s=4
t_{Bus}: Bus transfer	4	4	4	4	4	4	4
Intra-bank Constraints (cycle)							
t_{RL}: RD latency	6	8	9	9	11	12	18
t_{WL}: WR Latency	6	7	8	8	9	10	12
t_{WR}: WR data to PRE	8	10	12	12	14	16	12
t_{RCD}: Row to column delay	6	8	9	9	11	12	18
t_{RP}: Row precharge time	6	8	9	9	11	12	18
t_{RTP}: Column to precharge time	4	5	6	6	7	8	9
t_{RC}: Activation to activation	26	32	37	37	43	48	57
t_{RAS}: Row access strobe	20	24	28	28	32	36	39

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. Figure 2.4 depicts a simplified DRAM state diagram that shows the relationship and timing constraints between device states and commands. 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 [10, 117].

2.5 Related Work: Predictable Memory Controllers

Most COTS platforms deploy memory controllers with multiple architecture optimizations as an effort to overcome the effects of the interference on performance. These optimiza-

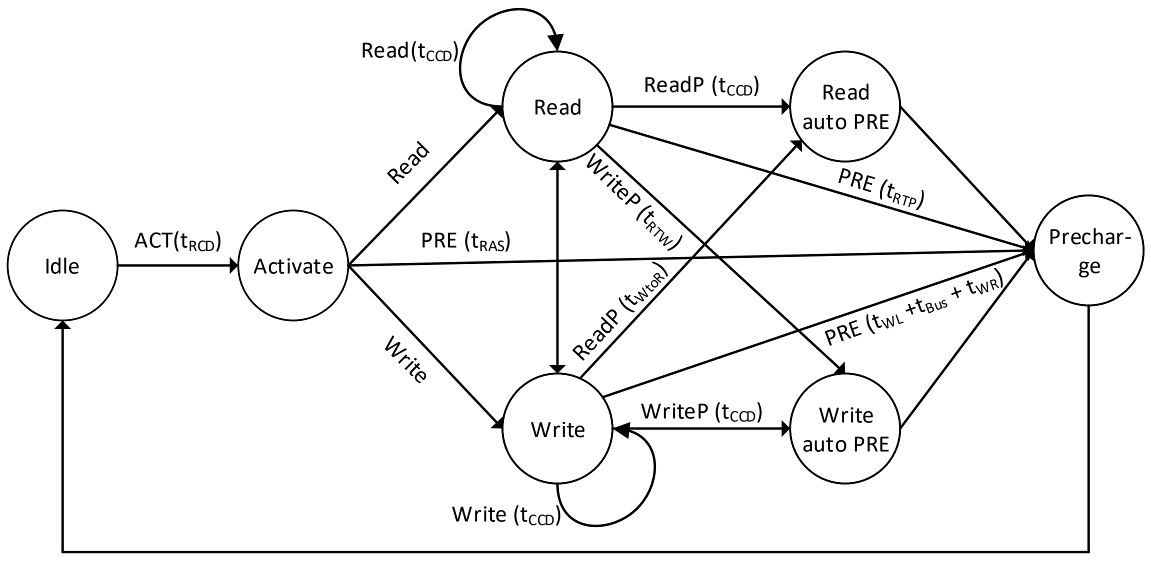


Figure 2.4: DRAM operation state machine diagram.

tions include exploiting locality through caching the most recently accessed DRAM page (known as open-page policy), reordering to prioritize requests to these cached pages (known as FR-FCFS scheduling), and read/write prioritization through write batching [121]. In particular, FR-FCFS arbitrates based on readiness and age of the requests. In FR-FCFS policy, assuming that there is no limitation on how many requests are re-ordered in the buffer, there can not be a bound on WCL of a request. FR-FCFS prioritizes memory requests that hit in the row-buffers of DRAM banks over other requests, including older ones. Row-buffer holds the most recently accessed row in that bank. If there is no request in the row buffer, then FR-FCFS prioritizes older requests similar to FCFS approach. This arbitration mechanism does not differentiate among requestors. Despite being beneficial for average performance, such optimizations were found to hurt analyzability and increase the WCL of DRAMs [44].

To overcome this challenge, real-time system researchers proposed to entirely redesign the memory controller to provide tight WCLs. Towards doing so, they disabled most aforementioned architectural optimizations. For instance, they prohibited the caching of DRAM pages by deploying a close-page policy, they replaced FR-FCFS with predictable scheduling such as RR or TDM, and they disabled write batching by equally scheduling reads and writes [2, 95, 22, 23, 24, 27, 59, 43]. As anticipated, these solutions successfully led to achieving better predictability for DRAMs; nonetheless, at the expense of significantly degrading average-case performance. This is ill-suited for modern embedded systems hosting mixed-criticality tasks with both latency-sensitive (critical) tasks and bandwidth-oriented (non-critical) tasks.

Accordingly, recent solutions targeted those mixed-criticality systems and offered compromise designs between the two extremes by adopting some of the architectural features existing in COTS platforms to enhance average-case performance, while deploying novel policies to still provide WCL bounds. We believe that this is a promising direction towards building high-performance yet predictable off-chip memory systems for mixed-criticality systems. Nonetheless, we believe the existing state-of-the-art works are still falling short to achieve this target. This is because when studying such works, we find them to suffer from one or more of the following drawbacks. 1) Some controllers favor performance at the expense of larger WCLs. For instance, controllers that deploy open-page policy (e.g. [116, 22, 23]) have different WCL scenarios based on the request type (whether page hit or miss, read or write), where a page miss suffers a significantly larger WCL (as large as the double based on our analysis). Although we believe the open-page policy is crucial for performance, such huge variability in the WCL based on the access behavior of the application hinders predictability; 2) On the other hand, other controllers aim at providing tighter bounds but suffer a substantial performance degradation [33, 84]. To exemplify, some solutions disable page caching by deploying close-page policy [33, 84], use static scheduling, or disable write batching, which results in large overhead delays of bus turnaround time upon switching the DRAM data bus direction between reads and writes [44]; 3) We find most of the existing proposals require drastic changes to the currently available COTS designs, which limits their applicability for the emerging markets of embedded systems such as automotive and avionics.

We can categorize the DRAM memory controller designs proposed in recent years into two categories; high-performance controllers and real-time controllers. Most high-performance controllers employ FR-FCFS arbitration to maximize bus utilization. In such controllers, the goal is to prioritize the row hit requests to increase the memory throughput. Although these approaches are in favor of the average-case performance, they lead to unfairness across different applications. For example, applications with a high hit ratio will be prioritized over other applications and impose considerable delay or even lead to starvation. There is much work explored to solve the problem mentioned above by modifying the memory controller architecture. Examples include [64, 79, 65, 107]. The key point in all of these works is to introduce application-aware scheduling. Controllers which employ FR-FCFS tend to re-order the requests. This behavior leads to unbounded latency, and the only possible approach to prevent it is limiting the number of re-ordering [62, 121] amongst requests. However, the WCL of such controllers can be very high or even unbounded, hence leading to poor predictability in the system. In terms of the average performance, generally, two reasons increase the WCL of a request when running alone. First, the delay from opening and closing rows, and second, switching the data bus direction. Data bus is bi-directional and requires a certain number of clock cycles to change the direction of CAS commands. Therefore, they reduce the bandwidth of the system. In a multi-core platform in which cores are sharing the banks, accessing different rows in the same bank reduces locality and this can happen frequently, thus, it increases the memory access latency and reduces the total bandwidth of the system. Note that periodic refreshes also take a significant amount of time, which leads to higher WCET [93, 21, 16].

There is a large body of research that focuses on real-time predictable memory controllers [84, 69, 42, 53, 59, 109, 116, 66, 22, 23, 24, 33, 85]. AMC [85, 84] employs static command scheduling with close-page policy to take benefits from constructing off-line command bundles for read/write requests. It also supports bank parallelism by interleaving operations of the same request over multiple banks. The command scheduler of AMC arbitrates between pending commands in FCFS manner. At the request scheduling level, AMC employs RR to provide fair arbitration among critical requestors and a lower priority assigned to non-critical requestors. PMC [43] employs a static command scheduling approach and introduces four static command bundles depending on the minimum request size in the system. Therefore, the page policy can be changed depending on the request size. The authors also proposed an optimization framework to generate an optimal work-conserving TDM schedule that supports mixed-criticality systems with different criticality requirements. Such requirements can be determined by maximum latency or minimum bandwidth for individual requestors.

RTMem [69] proposes a dynamic command scheduling with interleaved-banking and a hybrid page policy (which can be open or close). Dynamic command scheduling is used to improve average-case performance and allow for greater bank parallelism. Since RTMem is a back-end architecture, it can leverage any front-end request scheduler. RTMem considers variable request size by decoding each size into interleaved banks, and a number of read/write operations per bank. These parameters are selected offline to minimize the request latency. Similar to PMC, at run-time, RTMem issues open-row READ/WRITE commands until it reaches the last burst count, where auto PRE commands are issued to close the open rows. ORP [116] is the first approach in real-time memory controller literature that used open-page policy with dynamic command scheduling. In order to minimize the interference amongst the banks, private banking was used to avoid row interference. The memory controller uses RR arbitration for critical requests to provide the guarantee and also a complex FCFS arbitration among commands to take the benefit from the bank parallelism. ROC [66] improves over ORP by extending the DRAM module to multi-rank devices and mitigating the t_{WtoR} and t_{RTW} timing constraints since they do not apply between ranks. Hence, ROC provides two-level arbitration; the first level performs a RR among ranks, and the second level performs a RR among requestors assigned to banks in the same rank.

DCmc [53] also uses a dynamic command scheduler with an open-page policy and provides support for mixed-criticality and bank sharing among requestors. Critical requestors are scheduled based on RR in the same bank, while non-critical requestors are rescheduled according to FR-FCFS in the Soft Real-Time (SRT) banks to provide improved average-case performance. It also supports private banks per requestor and interleaved over shared banks. However, private banks are in favor of HRT tasks since they provide lower request latency bound. A more flexible bank assignment was proposed by MAG [114]. It assumes that SRT requestor can share the same bank with the HRT requestor but has lower priority during command scheduling and allows the preemption between SRT and HRT tasks. Since the write accesses are not on the critical execution path, MEDUSA [109] considers read access has higher priority than the write access. It assumes a write buffer with a fixed size and write accesses are only served if there is no read request or the write buffer reaches high watermark.

Mixed Critical Memory Controller (MCMC) [24] uses a similar rank-switching mechanism as in ROC but applies it to a simpler static command scheduling with a close-page policy. MCMC allows a predetermined number of critical and non-critical applications to coexist, while mitigation the memory interference for the critical applications. TDM arbitration is used to divide the timeline into a sequence of slots alternating between ranks. Each slot is assigned to a virtual device, and the HRT memory request has priority over the SRT requests. SRT requestors share the virtual device using round-robin arbitration. The SRT requestors receive non-predictable memory bandwidth since the slot can always be utilized by HRTs. As with TDM arbitration, the main drawback of this approach is that bandwidth will be wasted at run-time if no requestor is ready during a slot.

ReOrder [22] is also an open-page memory controller which improves over ORP by employing CAS reordering techniques and bundling read/write requests to reduce the access type switching delay. It uses a dynamic command scheduler among the three DRAM commands: RR for ACT and PRE commands, and read/write command reorder for the CAS command. Commands are issued in rounds where in each round only one turnaround in direction (read/write) is permitted. This eliminates repetitive CAS switching timing for read and write commands since as shown in [22], the switching delay between the different direction of CAS commands is the major timing constraint in the request latency. If a multi-rank module is in use, the controller schedules the same type of CAS in one rank, and then switches to another in order to minimize the rank switching as proposed in [23].

[33] proposed a new DRAM controller design named REQBundle that bundles memory requests of HRT applications in consecutive rounds according to their direction. This bundling based on the direction reduces the number of read-to-write and write-to-read switching delay. Open-page is applied for SRT requestors to increase bandwidth by allowing open requests to be serviced from row buffer and a close-page scheme is applied for HRT requestors to uphold predictability. In order to avoid process SRT and HRT applications at the same time, the controller works on the idea of a snapshot. When the snapshot is taken, the controller scans through the HRT bank command queues and accepts requests that are intra-ready. Any request that becomes intra-ready after this time will not be serviced in the current round. The request scheduler executes different arbitration policies for SRT and HRT banks. Employing request bundling with pipelining can improve the tightness of WCL because of increasing the number of commands that can be issued in a certain amount of time.

As it is clear in all the previous predictable memory controllers, there is a trade-off between average-case performance and predictable worst-case bounds, as techniques targeted at improving the former can harm the latter and vice-versa. We find that taking advantage of pipelining between different commands can improve both, but incorporating pipelining effects in worst-case analysis is challenging. To achieve this goal, we introduced a novel DRAM controller called DR*AMbulism* that successfully balances performance and predictability by employing a dynamic pipelining scheme [74]. In particular, our work show that the schedule of DRAM commands is akin to a two-stage two-mode pipeline. Therefore, we proposed an easily-implementable admission rule that allows the system to dynamically add requests to the pipeline without hurting worst-case bounds. Our evaluation shows that DR*AMbulism* provides comparable bounds to the most predictable real-time controller which is REQBundle [33] while delivering average performance similar to the highest-performance real-time controller that is ReOrder [22].

Another direction investigates the replacement of the commodity DDR DRAMs that have inherent big latency variations with different emerging memory types that are more predictable, such as the reduced latency DRAM (RLDRAM) [36]. Although such direction is achieving promising results, FCFS DRAMs are still the de facto standard for main memory, and hence, achieving predictability in systems adopting them remains an urgent challenge.

Chapter 3

Duetto Reference Model

In this chapter, we address the fundamental trade-off between average performance and predictability by introducing the Duetto reference model. Specifically, we focus on introducing Duetto general model on a simplified shared resource to exemplify how the approach works. In detail, we make the following contributions in this chapter.

1. We provide a conceptual description and formalization of the Duetto reference model in Section 3.2.

2. We exemplify the usage of Duetto to design a controller architecture in Section 3.3 for the case study of a system where cores share an interconnect to a shared multi-bank memory. The interconnect deploys a separate read and write bus connecting all cores to all memory banks and memory bus, which resembles the commodity buses existing in modern Systems-on-Chip (such as the ARM's AXI [4]).

3. Section 3.4 provides a detailed evaluation of the architecture in the case study by implementing the aforementioned resource and controller architecture in MacSim [60], a multi-core full-system, cycle-accurate simulator. Our results show that the derived architecture can achieve very close performance to a conventional high-performance arbiter while providing tight latency guarantees.

3.1 Case Study

We begin by describing the multi-bank memory used in our case study, so that we can use it as a running example throughout the chapter. We selected such resource since it allows us to highlight the key steps in the proposed design methodology, especially concerning parallelism in the hardware; at the same time, its behavior is not so complex as to prevent us from fully detailing the latency analysis in the available space. However, we have also validated the reference model on more complex memory models (specifically, DRAM in Chapter 5 and LLC in Chapter 4).

We consider a memory comprising N independent banks $b_1, ..., b_N$. All banks share one Read Bus (RB) and one Write Bus (WB). Each bank can only process either one read operation or one write operation at a time. A read operation requires t_r clock cycles to access the data in the bank, followed by t_{bus} cycles to transfer the data on the read bus. A write operation requires t_{bus} cycles on the write bus, followed by t_w clock cycles to store the data in the bank. A memory controller receives memory requests from cores, arbitrates among such requests, and sends read/write commands to the banks to trigger memory operations. The controller cannot issue a command to a bank b_j if the bank is busy processing a previous operation, or if doing so would create a bus conflict with the operation of a different bank. This implies that the controller can send at most one read and one write command to two different banks at the same time. Sending such commands takes one clock cycle. Note that split read/write buses are common in modern SoCs [103, 25, 102, 123] to reduce contention for access to memories and high-speed I/O; an example is the ARM AXI interconnect [4]. Compared to AXI, our model is slightly simplified, particularly in that we do not consider the time required for the cores to issue requests to the controller; however, this could be included by modeling the request bus and its associated arbiter as another resource, and then summing the two latencies over the request bus and the memory/data bus.

An example schedule is shown in Figure 3.1. We assume that the system is initially idle, then the following requests arrive at the controller: two read requests for bank b_1; one write request for b_2; and one read request for b_3. At time 0, the controller can send both a read command to b_1 and a write command to b_2. The controller must then wait until $t = 4$ to send the read command to b_3, since sending it earlier would cause a read bus conflict. Finally, to issue the second read command to b_1, the controller must wait until the latest of two events: the bank to become idle and for the absence of read bus conflict, which means the command is issued at $t = 8$.

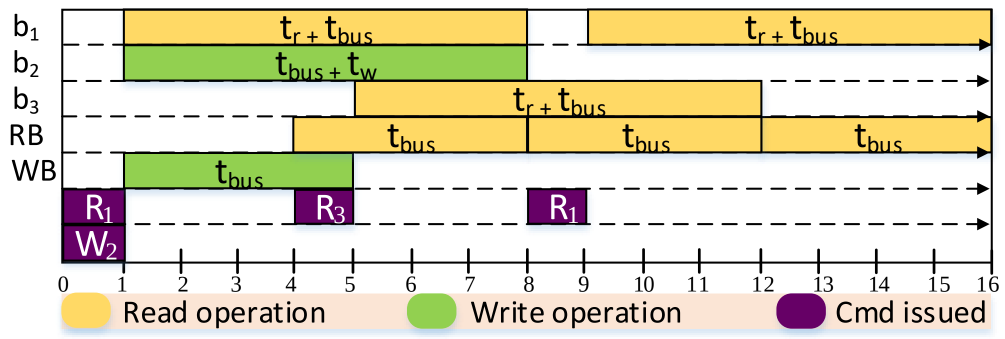


Figure 3.1: An example schedule for 4 requests accessing different banks with $t_{bus} = 4$ and $t_r = t_w = 3$.

3.2 Reference Model

We introduce the Duetto reference model. Concretely, we first decompose the system into a set of communicating conceptual components as shown in Figure 5.1. Then, we detail the execution model and discuss the provided latency guarantees.

3.2.1 Requestors and Requests

We assume that the system comprises M distinct *requestors*: $P_1, ..., P_M$, which issue *requests* for service to a shared *resource*. Depending on the resource, requestors could be cores, bus masters/DMA devices, or in general, any active hardware component. We assume that the requestors can have multiple outstanding requests. Upon being issued, a request is first stored in a *request buffer*; we call this the arrival time of the request. An *arbiter* then mediates access to the resource based on the stored requests. Finally, a request is removed from the request buffer once it completes service at the resource; we call this the finish time of the request. We assume that the requests of each requestor P_i can be totally ordered based on their arrival time, so that we can index them as $r_{i,1}, ..., r_{i,j}, ...$. Each request has a type; function $\mathcal{T}(r_{i,j})$ returns the type of $r_{i,j}$. We use R to denote the set of outstanding requests (requests that arrived and have not yet finished) at any one point in time; conceptually, this represents the state of the buffer. We do not impose any specific implementation of the buffer, albeit typically some form of hardware queue(s) is

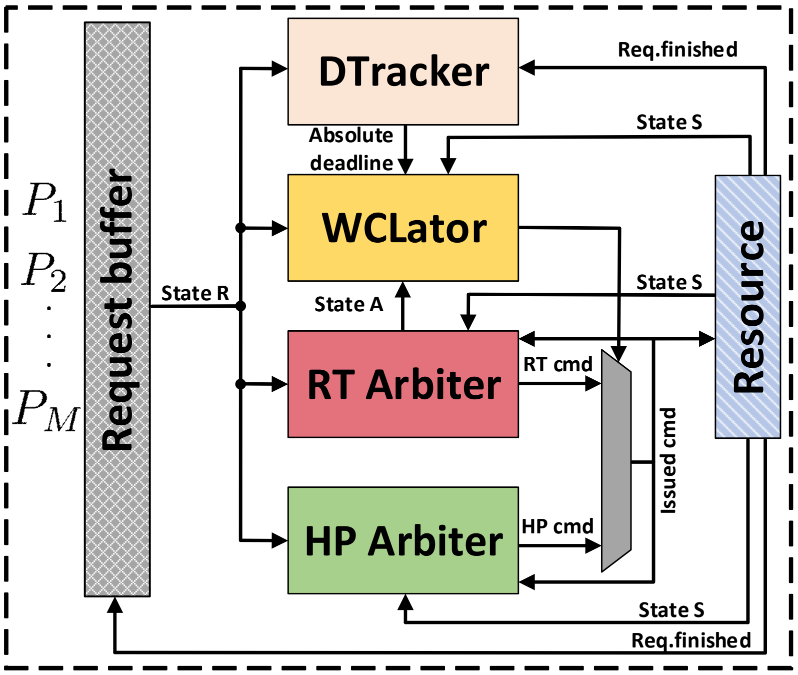


Figure 3.2: Duetto reference model.

employed. However, we assume that it is possible to construct a total order of the requests issued by each requestor based on the time at which they are issued.

Example: in our case study, requestors are Out-of-Order (OoO) cores. The type of a request is either read or write. Each core can issue multiple concurrent requests to any subset of banks. The considered arbiter resembles those in COTS by increasing parallelism through allowing to service requests OoO [97, 62]. For simplicity, we assume that a request finishes after the arbiter sends its read/write command.

3.2.2 Request Latency and DTracker

In real-time systems, timing guarantees depend on the WCET of tasks (schedulable entities). Following related work [34, 40], 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 bound on the total cumulative worst-case

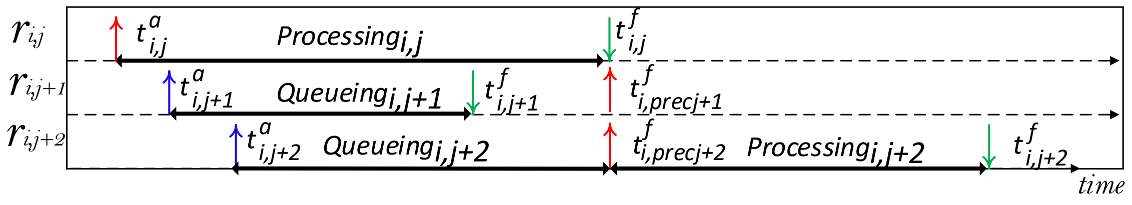


Figure 3.3: Queuing and processing latency example. Assume that all previous requests $r_{i,k}$ with $k < j$ finish before $t^a_{i,j}$. We use $\uparrow$ for the arrival time $t^a_{i,j}$ of each request, $\downarrow$ for its finish time $t^f_{i,j}$, and $\uparrow$ for the start of processing: $\max(t^f_{i,prec_j}, t^a_{i,j})$.

delay suffered by the task's requests to the shared resource under interference. $\mathcal{D}_i$ is calculated as the bound on the worst-case number of requests multiplied by the worst-case processing latency of each request [45]. In a multitasking system, the same approach can be applied [45, 62]: 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. For example, response time analysis can be used to determine which tasks execute in a given busy interval, what is their pure computation time and how many requests they generate per-core [45, 62]. The length of the busy interval is then obtained by summing the computation time of the tasks plus the overall request delay.

The goal of Duetto is to provide worst-case guarantees on the latency of each request. Since a requestor might have multiple outstanding requests (for instance OoO cores, superscalar, GPU, etc.), requests might not be serviced in order. Therefore, it is necessary to precisely define the concept of latency. Following related work [62, 71], we use the following definition:

Definition 1 (Queuing and Processing Latency). *Let $t^a_{i,j}$ be the arrival time of request $r_{i,j}$, let $t^f_{i,j}$ be its finish time, and 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* queuing latency *of $r_{i,j}$ is* $\max\left(0, \min(t^f_{i,j}, t^f_{i,prec_j}) - t^a_{i,j}\right)$, *while the* processing latency *is* $\max\left(0, t^f_{i,j} - \max(t^f_{i,prec_j}, t^a_{i,j})\right)$.

Figure 3.3 shows an example with three requests. In this figure, since $t^f_{i,prec_j} < t^a_{i,j}$, the queuing latency of $r_{i,j}$ is zero meaning that while $r_{i,j}$ is being serviced, there is no other request processing in the resource. As in all related work, our reference model bounds the processing latency only. The key reason is that whenever a requestor issues multiple requests and stalls until they complete, the stall time is upper bounded by the sum of the processing delay of the requests. In particular, as discussed in [115], this means that the delay suffered by a real-time task executed on a core accessing a shared resource can be bounded by the sum of the processing delay of the requests issued by the task. For this reason in Figure 3.3, the processing latency of $r_{i,j+1}$, which is covered by the processing time of $r_{i,j}$, is set to zero. Furthermore, the processing time of $r_{i,j+2}$ only starts when the processing of previous request to the resource is completed ($r_{i,j}$) and while $r_{i,j+2}$ is waiting to start the processing, it is suffering a queuing delay as shown in the figure.

Based on the above discussion, latency requirements in our reference model are expressed by associating each requestor P_i and each type of request with a *relative deadline* $D_i\big(\mathcal{T}(r_{i,j})\big)$, which represents the maximum allowable processing latency for each request of that requestor and type. Consequently, the finishing time of every request $r_{i,j}$ must be no later than its *absolute deadline* $d_{i,j} = \max(t^f_{i,prec_j}, t^a_{i,j}) + D_i\big(\mathcal{T}(r_{i,j})\big)$. The *Deadline Tracker* (DTracker in Figure 3.2) component is responsible for maintaining such information using a *slack counter*, which counts the number of clock cycles until the absolute deadline (i.e., at clock cycle t, the value of the slack counter is $\max(t^f_{i,prec_j}, t^a_{i,j}) + D_i\big(\mathcal{T}(r_{i,j})\big) - t$). Note that it suffices to maintain a single absolute deadline for each requestor P_i, associated with its oldest outstanding request $r_{i,j}$: this is because subsequent requests of P_i cannot have an absolute deadline earlier than the finish time of $r_{i,j}$. To simplify exposition, in the rest of the chapter we will thus use the term "oldest request" to denote any request that is the oldest for its requestor (rather than the oldest among all requestors).

Definition 2 (Oldest Request)**.** *At any time t, the oldest request of a requestor (if any) is the earliest arrived request of that requestor that is still outstanding at t.*

3.2.3 Commands and Resource Interface

An arbiter controls the resource to service requests by issuing one *command* every clock cycle. Such commands alter the internal state s of the resource. We use $\mathcal{S}$ to denote the set of all possible resource states. Like requests, every command is characterized by a type. A "no-operation" NOP command is used whenever the resource is idle.

Example: based on the discussion in Section 3.1, the resource accepts four types of commands: NOP, RD, WR, and RD/WR. Note that while in this case a request is serviced by a single command, depending on the resource, additional commands might be required. For example, a DRAM request might require a PRE, ACT, and CAS commands [106].

We assume that the command semantic is defined by an automata, which we call the *resource interface*. Essentially, the interface defines the "contract" between the resource and the arbiter; in this sense, it does not need to model the low-level internal state of the resource, but rather only those details that are relevant in terms of the behavior of the commands. For many resource types, the interface is defined by a standard, e.g. JEDEC for DRAM [106]. Let S to denote the current state of the resource interface; we say that a command is *valid* in S if the resource can accept that command. We further say that a command is *legal* if it is valid and satisfies an outstanding request in R, and use $\mathcal{L}(S, R)$ to denote the set of legal commands. Note that a command might be valid but not legal: for example, a requestor might issue a read request to a memory resource, and an erroneous arbiter might then generate a valid but incorrect write command for that request.

The number of command types (other than NOP), as well as the number of commands that must be issued to satisfy a given request, depend on the resource. For example, in the case of a simple bus, the only operational command might be $grant(i, t)$, granting access to the bus to P_i for t clock cycles. For a read request in data cache resulting in a miss, operations might include reading the tag memory, sending a fetch request to main memory, evicting a cache line, and more if coherency is supported.

Example: the behavior of the resource interface can be defined using $N + 2$ timers: c^r, c^w for the read and write bus, and c^b_j for each bank b_j. c^r (c^w) is set to t_{bus} every time a RD (WR) command or RD/WR command is issued. Whenever a command is sent to bank b_j, c^b_j is set to either $t_r + t_{bus}$ (for a read operation) or $t_{bus} + t_w$ (for a write). A command is valid only if all relevant bank and bus timers are zero. We say that a request is ready if it can be serviced by issuing a legal command in the current clock cycle.

3.2.4 High-Performance and Real-Time Arbiter

The reference model comprises two arbiters: a HPA, which we assume to be optimized for maximum average performance, and a RTA, optimized for tight latency bounds. Since this chapter focuses on timing requirements, and not functional verification, for simplicity, we will assume that the HPA is *correct*, in the sense that it always issues legal commands. If the correctness of the HPA cannot be verified, the methodology can be extended with an additional checker module that checks the legality of the commands based on the states

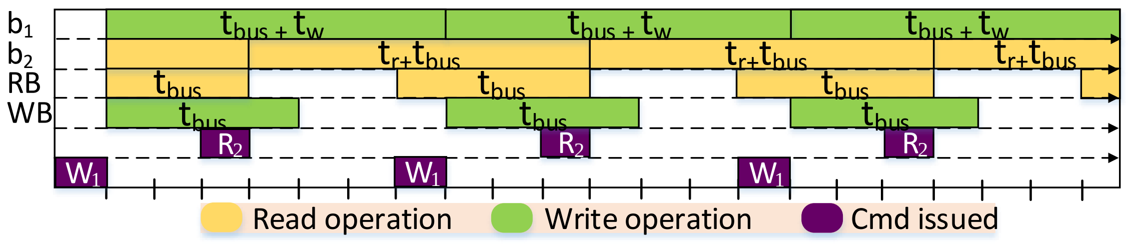


Figure 3.4: Unbounded latency. A read request to b_1 is never ready, since the read bus is occupied whenever b_1 becomes idle.

S, R of the resource interface and request buffer. We do not make any further assumptions on the way that requests and their corresponding commands are scheduled by the HPA. In other words, we do not need to analyze the behavior of the HPA, and its behavior can be modified without impacting real-time guarantees. This ensures that the latency guarantees provided by our methodology are completely independent of the HPA. On the other hand, the behavior and internal state of the RTA, which we denote as A, must be explicitly modeled.

Example: for our evaluation in Section 3.4, we employ a FR-FCFS arbiter as the HPA. At each clock cycle, this arbiter selects among ready requests, and gives priority to requests based on their arrival time. We choose this arbiter because, as shown in literature [97], it tends to maximize performance in terms of the overall Instructions Per Cycle (IPC) of the system by favoring applications with high IPC that can issue multiple concurrent memory requests. However, note that this arbiter does not provide any latency guarantee. In particular, as shown in Figure 3.4, we can construct a pattern of memory accesses where a read request to bank b_1 can be stalled for any amount of time by a sequence of write requests to b_1 and read requests to other banks. We discuss the design of the RTA in Section 3.3.

3.2.5 Execution Model and Latency Guarantees

Finally, we discuss the execution model and how latency requirements are guaranteed by the Worst-Case Latency Estimator (WCLator) component, which is the core of our reference model. As Figure 3.2 illustrates, the two arbiters operate independently and in parallel. Every clock cycle, each arbiter selects one command (possibly NOP) based on its internal state, as well as the states S, R of the resource interface and request buffer. In parallel, the COTS selects between the two arbiters; the command of the selected arbiter is then issued to the resource through the multiplexer in the figure. Since the command issued in a clock cycle might be different from the one selected by an arbiter, we require that each arbiter updates its internal state based on the actual issued command, rather than the one it selects. It is essential to note that because we are targeting high-speed hardware implementation, we assume that **the WCLator must make its decision without knowing which commands are selected by the two arbiters**: otherwise, the WCLator logic would have to be placed in series with the arbiters, which could greatly slow down the clock speed.

To decide between the arbiters, at each clock cycle and for each requestor P_i with one or more outstanding requests, the WCLator computes an upper bound to the finish time $t^f_{i,j}$ of its oldest request $r_{i,j}$, under the assumption that **any legal command can be sent in the current cycle, while the RTA is selected in all future cycles.** If for each such requestor, the computed finish time is less than or equal to the deadline $d_{i,j}$, then the WCLator selects the HPA. Otherwise, it selects the RTA. The intuition for this decision is that if the computed finish time is no larger than the deadline, then it is safe to continue with the (legal) command that is selected by the HPA given that the WCLator can switch to the RTA from the following cycle. Note that the WCLator's estimation can be based on the current states S, R and A of the resource interface, request buffer and RTA. We next formally prove that the system meets all deadlines, as long as the latency requirements are not set to a smaller value than the latency guarantees provided by the RTA.

Definition 3 (Static Worst-Case Latency (WCL) Bound)**.** *For every requestor P_i and request type, let $\Delta_i\big(\mathcal{T}(r_{i,j})\big)$ to be an upper bound to the processing latency of $r_{i,j}$ assuming any possible state of the resource interface S, request buffer R and RTA A at time $\max(t^f_{i,prec_j}, t^a_{i,j})$, and that the WCLator always selects the RTA from $\max(t^f_{i,prec_j}, t^a_{i,j})$ onward.*

Theorem 4. *No request misses its deadline if for all requestors and request types it holds: $D_i\big(\mathcal{T}(r_{i,j})\big) \geq \Delta_i\big(\mathcal{T}(r_{i,j})\big)$.*

Proof. By contradiction, assume there exists a request $r_{i,j}$ that misses its deadline at $d_{i,j}$. Since $d_{i,j} = \max(t^f_{i,prec_j}, t^a_{i,j}) + D_i\big(\mathcal{T}(r_{i,j})\big)$ and no older request of P_i can finish after $t^f_{i,prec_j}$, it follows that $r_{i,j}$ is the oldest request of P_i in the interval $[\max(t^f_{i,prec_j}, t^a_{i,j}), d_{i,j})$. We consider two cases: (1) the WCLator always sends commands of RTA in $[\max(t^f_{i,prec_j}, t^a_{i,j}), d_{i,j})$; (2) or the WCLator sends at least one command of HPA during such interval; in which case let t be the latest time at which an HPA command is sent.

Case (1): by Definition 3 and since the RTA is always selected, $r_{i,j}$ must finish by $\max(t^f_{i,prec_j}, t^a_{i,j}) + \Delta_i\big(\mathcal{T}(r_{i,j})\big) \leq \max(t^f_{i,prec_j}, t^a_{i,j}) + D_i\big(\mathcal{T}(r_{i,j})\big) = d_{i,j}$; a contradiction.

Case (2): since the WCLator selects the HPA at time t, while the RTA is selected for all following cycles until $d_{i,j}$, and because by assumption the HPA is correct, it follows that $t^f_{i,j} \leq d_{i,j}$, again a contradiction. □

The key intuition behind Duetto is that using run-time information on the state of the system typically allows the WCLator estimation to be much tighter than any possible static WCL bound. Hence, unless the system becomes fully loaded, the WCLator can keep selecting the HPA and avoid loss of average performance. It is important to point out that S consists of any possible states and not just the states that are reachable by the RTA. In detail, at any point in time, the HPA can be selected and potentially bring the system to different states than the one that is reachable by RTA. For this reason, the analysis cannot be done assuming that the RTA always running from the beginning of the execution. This might be slightly pessimistic compared to assuming that RTA runs forever. However, in terms of deriving the analysis bound, the operation of HPA does not matter since the bound is computed assuming that the RTA always runs.

3.3 Architecture Design

We next show how to employ the reference model to design a concrete architecture. We consider a use-case where the resource, HPA and request buffer have already been designed/implemented, and the designer wishes to add support for latency requirements using Duetto. The detailed DTracker design depends on the request buffer, but we argue that it is generally straightforward [1]. Therefore, we focus on the design of the RTA and WCLator. We do not claim that an automated process is possible; however, we believe that the design can proceed through a sequence of four conceptual steps, which we illustrate based on our case study.

[1]To simplify implementation, it is preferable to store the number of cycles remaining until the absolute deadline, rather than the absolute deadline itself.

Table 3.1: Symbols used in latency analysis.

Symbols	Description
P_i	Core i
b_k	Bank k
hp_i	Set of higher priority requestors than P_i
t_{bus}	Clock cycles to transfer the data on read/write bus
t_r	Clock cycles to access the data in the bank
t_w	Clock cycles to store the data in the bank
c^r	Counter for read bus
c^w	Counter for write bus
c_j^b	Counter for bank j
$k^{bank,r}$	Number of read oldest requests of requestors in hp_i that target b_k
$k^{bank,w}$	Number of write oldest requests of requestors in hp_i that target b_k
$k^{bus,r}$	Number of oldest read requests of requestors in hp_i that target another bank
$k^{bus,w}$	Number of oldest write requests of requestors in hp_i that target another bank

3.3.1 Step A: RTA Design

We design a dynamic RR arbiter, which provides the same latency guarantees to every requestor without unduly limiting bank parallelism. A requestor is removed from the RR queue when its oldest request finishes, and enqueued at the back of the RR queue if it has any outstanding request. For a requestor P_i, we use hp_i to denote the set of higher priority requestors, i.e. the requestors that are ahead of P_i in the RR queue. We say that a request of P_i to bank b_k is blocked if there is a non-ready oldest request of a requestor in hp_i that also targets b_k. The RTA arbitrates between non-blocked ready requests based on a two-level arbitration scheme:

1. In the first level, it gives priority to oldest requests over non-oldest requests.
2. In the second level, it uses the RR order of requestors.

This means that if, for example, the oldest request of the highest priority requestor is non-ready because its bank is busy, the controller can still service a lower priority request to another bank. However, if the highest priority request is non-ready because its data bus is busy, the controller cannot service a lower priority request of the opposite type (read to write or vice-versa) to the same bank, since this could result in the pattern in Figure 3.4.

3.3.2 Step B: Dynamic RTA Latency Analysis

We compute an upper bound to the remaining latency (i.e., the time to finish) of the oldest request $r_{i,j}$ of P_i assuming that the RTA is always selected. We encode the states S, R and A into a small set of *analysis parameters* used to derive latency equations. In this chapter, we only consider the case of a read request, but the write case is similar. Let b_k be the bank targeted by $r_{i,j}$. We use $k^{bank,r}, k^{bank,w}$ to denote the number of read/write oldest requests of requestors in hp_i that target b_k, and $k^{bus,r}, k^{bus,w}$ to denote the number of oldest requests of requestors in hp_i that target another bank; note that such parameters can be easily derived at run-time based on the state of the request buffer R and RR queue in the RTA (A). Table 3.1 summarizes the symbols used in the latency analysis.

Theorem 5. *If the oldest request of P_i at time t is read request $r_{i,j}$ targeting b_k, and the RTA is always selected from t onward, its remaining latency $t^f_{i,j} - t$ is bounded by:*

$$\text{if } k^{bank,w} = 0 : c^{init,r} + k^{bank,r} \cdot (t_r + 2 \cdot t_{bus} - 1) + k^{bus,r} \cdot t_{bus} + 1, \tag{3.1}$$

$$\begin{aligned} \text{if } k^{bank,w} > 0 : \quad & c^{init,rw} + k^{bank,r} \cdot (t_r + 2 \cdot t_{bus} - 1) + \\ & k^{bank,w} \cdot (t_w + 2 \cdot t_{bus} - 1) + \\ & (k^{bus,r} + k^{bus,w}) \cdot t_{bus} + 1, \end{aligned} \tag{3.2}$$

where:

$$\begin{aligned} \text{if } c^r \geq c^b_k : \quad & c^{init,r} = c^r, & (3.3)\\ \text{if } c^r < c^b_k : \quad & c^{init,r} = c^b_k + t_{bus} - 1, & (3.4)\\ \text{if } c^r \geq c^b_k \wedge c^w \geq c^b_k : \quad & c^{init,rw} = \max(c^r, c^w), & (3.5)\\ \text{if } c^r < c^b_k \vee c^w < c^b_k : \quad & c^{init,rw} = c^b_k + t_{bus} - 1. & (3.6) \end{aligned}$$

Proof. Since oldest ready requests are arbitrated in RR order, and requests to b_k are blocked if there is a higher priority non-ready request, it follows that the RTA will service a sequence of exactly $k^{bank,r} + k^{bank,w}$ requests to b_k followed by $r_{i,j}$ itself. Furthermore, after a request in the sequence becomes ready and non-blocked, it can still be delayed by higher priority requests targeting the same bus but a different bank; if $k^{bank,w} > 0$, then conflicts can happen over both the read and write bus; hence we consider $k^{bus,r} + k^{bus,w}$ conflicting requests, otherwise ($k^{bank,w} = 0$), we only consider the $k^{bus,r}$ read requests. Then, the remaining latency of $r_{i,j}$ can be obtained by summing: (1) the time until the first request in the sequence to b_k first becomes ready; we call this either $c^{init,r}$ (if $k^{bank,w} = 0$) or $c^{init,rw}$ (if $k^{bank,w} > 0$). (2) The latency between issuing the command for one request in the sequence, and the time the next request in the sequence becomes ready. Each request in the sequence occupies b_k for either $t_r + t_{bus}$ (read) or $t_{bus} + t_w$ (write) cycles; furthermore, a lower priority request could be serviced the cycle before the bank becomes idle, adding an extra $t_{bus} - 1$ cycles of delay. Hence, the overall latency over the sequence is equal to: $k^{bank,r} \cdot (t_r + 2 \cdot t_{bus} - 1) + k^{bank,w} \cdot (t_w + 2 \cdot t_{bus} - 1)$. (3) The delay of higher priority requests targeting a different bank; as argued, this is $(k^{bus,r} + k^{bus,w}) \cdot t_{bus}$ if $k^{bank,w} > 0$, and $k^{bus,r} \cdot t_{bus}$ otherwise. (4) One clock cycle to issue a RD or RD/WR command to service $r_{i,j}$. Summing the four terms yields Equations 3.1, 3.2.

Finally, we consider $c^{init,r}, c^{init,rw}$. As shown in Equations 3.3-3.6, the two cases differ only in which bus timers we need to consider, based on the type of requests in the sequence to b_k. If at time t, the relevant bus timer(s) is larger or equal than the bank timer c_k^b, then the first request in the sequence will become ready when the bus timer(s) expire. Otherwise, it is again possible for a lower priority request to be serviced one cycle before b_k becomes idle, resulting in an initial delay of $c_k^b + t_{bus} - 1$. This concludes the proof. □

3.3.3 Step C: Static WCL Bound

The static WCL bound $\Delta_i\big(\mathcal{T}(r_{i,j})\big)$ is obtained by maximizing the remaining latency in Equations 3.1-3.6 over all possible values of the parameters. Specifically, we set c_k^b to its maximum value $\max(t_r, t_w) + t_{bus} - 1$ (a request was issued to b_k the previous cycle), and set $k^{bank,r} + k^{bank,w} = M - 1$, $k^{bus,r} = k^{bus,w} = 0$ (requests to bank b_k generate larger delay, and there can only be one oldest request for each of the $M - 1$ other requestors), yielding:

$$\Delta_i(\text{read}) = M \cdot \big(\max(t_r, t_w) + 2 \cdot t_{bus} - 1\big). \tag{3.7}$$

Repeating the analysis for a write request yields the same bound. Hence, in our example, all request types have the same static WCL bound; and since we treat all requestors equally, the bound does not depend on the requestor either. However, more complex arbitration schemes could differentiate between request types [121, 44] or requestors [46, 40] based on criticalities. The obtained bound is predictable, in the sense that it is linear in the number of requestors; each requestor contributes a latency term $\max(t_r, t_w) + t_{bus}$, which represents the worst-case intra-bank time, plus a blocking term $t_{bus} - 1$, which represents the price for allowing inter-bank parallelism.

3.3.4 Step D: WCLator Design

Consider again the oldest request $r_{i,j}$ of P_i targeting bank b_k at time t. To estimate its finish time $t^f_{i,j}$, we enumerate a set of cases based on which request(s) (at most one read and one write) could be serviced by a legal command issued at time t. For each case, we compute a bound to the remaining latency of $r_{i,j}$, so that we can obtain $t^f_{i,j}$ by summing the remaining latency with t.

- **(1)** If $r_{i,j}$ is serviced, then its remaining latency is 1 cycle.
- **(2)** Otherwise, the remaining latency is computed by using Equations 3.1-3.6 but with a modified value of some parameters, as detailed in the sub-cases below, to account for the command sent at t; this is possible because by definition the COTS computes $t^f_{i,j}$ assuming that the RTA is selected from $t + 1$ onward.
- **(2.1)** If a request to b_k is serviced, set $c^b_k = t_r + t_{bus}$ or $c^b_k = t_{bus} + t_w$, depending on the type of request; and subtract one from $k^{bank,r}$ or $k^{bank,w}$ if it is the oldest request of a requestor in hp_i.
- **(2.2)** If a request to another bank is serviced, set either $c^r = t_{bus}$ or $c^w = t_{bus}$, depending on the type of request; and subtract one from $k^{bus,r}$ or $k^{bus,w}$ if it is the oldest request of a requestor in hp_i.
- **(2.3)** If neither sub-case (2.1) or (2.2) apply, then a *NOP* is issued. In this case, no change is needed if the value of $c^{init,r}$ in Equation 3.1, or the value of $c^{init,rw}$ in Equation 3.2, is greater than zero; otherwise, add one to the latency computed by the equation to account for the cycle wasted by issuing the *NOP*.

At run-time and for each requestor P_i, the COTS then uses the set of legal commands $\mathcal{L}(S, R)$ to determine which cases can apply; and takes $t^f_{i,j}$ as the maximum over all such cases. Here, sub-case (2.1) can be excluded if $c_k^b > 0$, or there is no request targeting b_k apart from $r_{i,j}$ in the request buffer; combining such information with the state of the RR queue in the RTA allows the COTS to also determine whether $k^{bank,r}$ or $k^{bank,w}$ should be decreased or not (note that if possible, the latter case must be considered since it leads to higher latency). Similar considerations hold for sub-case (2.2). The presented design is well-suited for hardware implementation, because each case for every requestor can be computed in parallel [2], and the resulting $t_{i,j}^f$ compared against $d_{i,j}$ to determine if the HPA can be selected. Hence, the complexity of the implementation depends on the latency equations. As shown by Equations 3.1-3.2, we argue that most analysis for predictable arbiters [119, 40, 62, 121, 44] yield equations that involve adding terms, where each term depends on an analysis parameter. This can be computed efficiently in hardware by using one look-up table for each term, and then cascading the results through a sequence of adders.

Note that it is not important who the requestor is if a request is not the oldest of a higher priority one, since in that case P_i will keep its position in the RR queue. Also note that subcases 2.2 can be excluded if $c_j^b > 0$, or there is no other request to bank j apart from r in the request buffer; while subcases 2.3 and 2.4 can be excluded if $c^r > 0, c^w > 0$, or there is no read/write request apart from r in the request buffer (similarly, we can decide whether to subtract or not from the k values based on which requests are in the global request buffer).

3.4 Evaluation

We use MacSim [60], an x86 multi-processor architectural simulator, to model the requestors in our evaluation. We incorporate eight 8-wide superscalars (i.e. it can process multiple instructions per cycle) cores clocked at 1GHz and implement the case study in the open-source MCsim memory controller simulator [73][3]. All cores share a bus connected to the multi-banked memory, as explained in the case-study throughout the chapter. We employ two types of synthetic benchmarks: latency-sensitive and bandwidth-oriented from IsolBench [110]. We run the non memory-intensive benchmark on the foreground core and memory-intensive benchmarks on the background cores. Memory requests are interleaved among all banks at the granularity of a cache line (64 bytes).

[2]To reduce area, an optimized implementation can merge cases that have similar bounds and remove those with provably smaller latency.

[3]We introduce MCsim thoroughly in Chapter 5

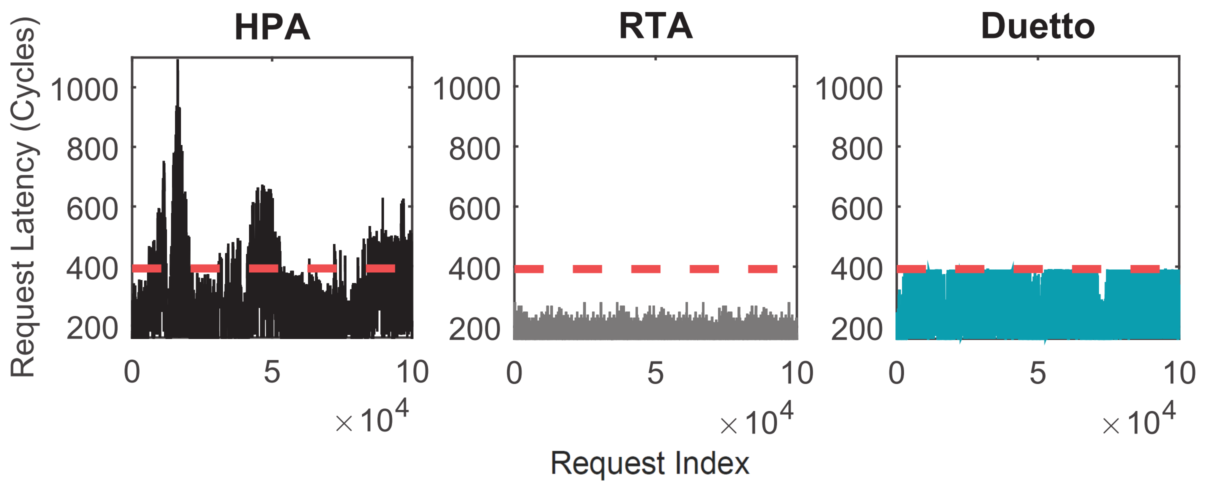
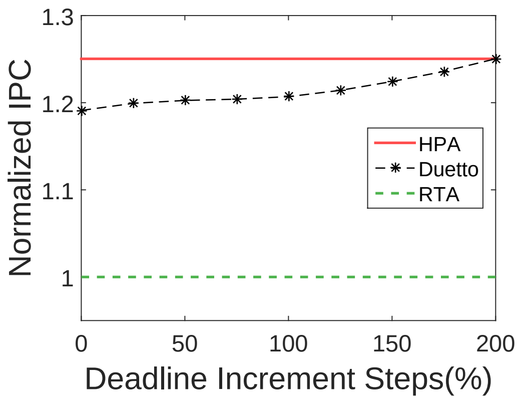


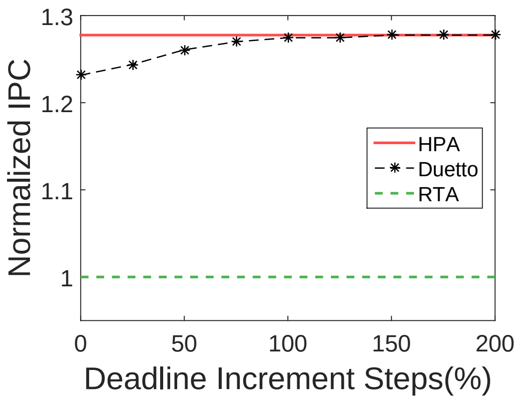
Figure 3.5: Request latency of HPA, RTA, and Duetto.

Figure 3.5 delineates the processing latency of each request under HPA, RTA and Duetto when all requestors contend for access to $N = 8$ banks with a bus time $t_{bus} = 10$ cycles and processing time $t_r = t_w = 30$. For visualization reasons, the figure only incorporates requests with latency longer than 150 cycles. The red line represents the static WCL bound. For HPA, we observe large latency spikes throughout the execution (the maximum observed latency is 1094 cycles), which shows 179% increase compared to the static bound. This is because HPA prioritizes requests that target a ready bank, which can starve (theoretically) or delay for a long time (practically) requests targeting busy banks. On the other hand, RTA guarantees the latency bound for all requests as expected. However, none of the requests come close to the static WCL bound (392 cycles): this is because the static analysis must assume that all requestors access the same bank at the same time, which is unlikely in practice.

Finally, for Duetto we used the minimum possible relative deadline $D_i\big(\mathcal{T}(r_{i,j})\big) = \Delta_i\big(\mathcal{T}(r_{i,j})\big)$ for each requestor. The deadline value for each requestor is configurable and can be determined based on the requirements and characteristics of the application/requestor. Duetto stretches the latency of requests towards the relative deadline (red line), allowing it to keep selecting the HPA as long as possible. which is due to the fact that the COTS estimates the latencies at run-time, and has thus more information on the requestors and banks (e.g., RR priorities). This allows Duetto to keep selecting the HPA as long as no request risks violating its deadline, thus significantly improving average per-



(a) IPC ($t_r = t_w = 0$).



(b) IPC ($t_r = t_w = 30$).

Figure 3.6: IPC evaluation of two different configurations under Duetto.

formance compared to RTA as we show next. Notice that Figure 3.5 does not reflect all requests as they only represent oldest requests with latencies greater than 150 cycles. The average request latency among all requests in HPA is lower than Duetto and consequently both HPA and Duetto are lower than RTA.

We use the aggregate IPC of the workload over 8 cores as a measure of performance. Figures 3.6a and 3.6b show the IPC of RTA, HPA and Duetto normalized by the IPC of RTA, when setting either $t_r = t_w = 0$ or $t_r = t_w = 30$. Notice that, $t_r = t_w = 0$ implies the requestors only compete to access the read/write buses, i.e. there is no bank parallelism. For Duetto we first set all deadlines to the minimum possible value, and then progressively increase them up to $3 \times \Delta_i$ (200% increase). From the figures, the performance of Duetto is already close to HPA with the strictest latency requirements, and relaxing such requirements further increases its performance until it matches the HPA. Furthermore, bank parallelism improves the relative performance of Duetto, as it increases the difference between the static WCL bound and the dynamic bound. We have also experimented by changing the number of banks and the way that requests are interleaved, but we omit such results as they show little variations in terms of the relative performance between HPA, RTA, and Duetto.

3.5 Summary

We introduced the Duetto reference model, a novel paradigm for shared hardware resource management in real-time embedded systems. By pairing a high-performance COTS arbiter with a predictable real-time arbiter and dynamically switching between the two at run-time, Duetto is able to overcome the traditional trade-off between average-case performance and predictability. Notice that, in this chapter we decided to consider all the requestors with same priority to give them same latency bounds. Without this consideration, and similar to other proposals in the literature [40, 24], we could make some requestors strictly lower priority and this would result in better static bound for the higher priority requestors. We next demonstrate Duetto on a broad spectrum of resources, including bus/LLC and DRAM in Chapter 4 and Chapter 5.

Chapter 4

DUEPCO: Applying Duetto to Cache Coherency with Added Parallelism

In this chapter, we propose a solution aiming at providing predictable, coherent shared cache hierarchy solution, yet with a negligible performance degradation compared to COTS solutions. This goal is achieved by adopting a high-performance-driven architecture including a split-transaction bus and proposing to bankize the shared caches to resemble a system with multiple shared resources. In addition, all accesses are arbitrated through a global ordering mechanism. Our proposed arbiter operates alongside conventional coherence protocols without requiring any protocol modifications. Furthermore, this chapter applies Duetto reference model on the hardware cache coherency.

Originally designed with performance as the main goal, COTS cache coherent interconnects deploy several re-orderings and optimizations that hinder their predictability. It has been shown that even deploying a simple COTS coherence protocol such as the MSI protocol on top of a TDM-based interconnect revokes the system predictability [39, 55]. To address this problem, most existing solutions aiming to provide predictable cache coherence impose both coherence protocol as well as architectural modifications [39, 104, 57, 55] or at the very least require specific hardware support [38]. These changes have led in early works to a quadratic increase in the worst-case memory latency due to coherence [39, 104, 57] (***Problem ❶***). Moreover, mandating coherence protocol modifications discourages a real adoption of these solutions from the industry since adoption and verification of a new coherence protocol is known to be one of the most complex architectural tasks [5, 103, 90] ***(Problem ❷)***. PISCOT [49] addresses these problems by enabling the deployment of COTS coherence protocols on split-transaction interconnects. PISCOT also reduces the quadratic worst-case coherence latency to be linear in the number of cores. Nonetheless, PISCOT achieves this tight WCL by deploying two techniques that limit the overall memory performance compared to COTS solutions. The first is a TDM-based request bus, which is needed to enforce predictability, and the second is limiting the number of requests that each core can issue to the interconnect to one, which is needed to achieve the aforementioned tight latency ***(Problem ❸)***. Additionally, similar to all existing work, PISCOT models the data bus and the LLC as a single shared resource, and hence, no parallelism is possible in accessing the LLC ***(Problem ❹)***. In COTS platforms, since bank processing times are much longer than the data transfer on the bus, LLC is usually a bankized memory, where different banks can process requests in parallel to improve the system's performance [50]. Motivated by these limitations in the state-of-the-art works, this chapter makes the following contributions:

1. We propose a novel real-time arbitration scheme for managing memory accesses in the cache hierarchy. This arbiter models cache hierarchy as independent and parallel resources: the request (control) bus, the response (data) bus, while each LLC's bank is a resource of its own. This is key to leverage parallelism among these components to improve average performance, while tightening memory latency bounds (addressing ***Problems ❶*** and ***❹)***. More details about this arbiter are in Section 4.2.

2. To further address the performance-predictability trade-off in modern embedded systems, we propose DUEPCO that applies the Duetto reference model on the hardware cache coherency. This is achieved by integrating two arbiters: a High-performance Arbiter (HPA) that offers the system a COTS-level performance most of the time, while the proposed Real-time Arbiter (RTA) runs in parallel and is only utilized when necessary to meet timing guarantees (addressing ***Problem ❸)***. Section 4.4 discusses the operation of DUEPCO.

3. We provide a timing analysis that ensures predictability by statically bounding the worst-case latency suffered by any memory request. Unlike the solutions in [39, 55, 57, 104], and similar to [49, 38, 56], this bound is linear in the number of cores (addressing ***Problem ❶***). Furthermore, DUEPCO is also able to track the dynamic latency behavior of memory requests at run-time, which enables us to further the decision of which arbiter to be used (real-time or high-performance) based on the current status of the resources and the pending requests from different cores. This is discussed in detail in Section 4.3.

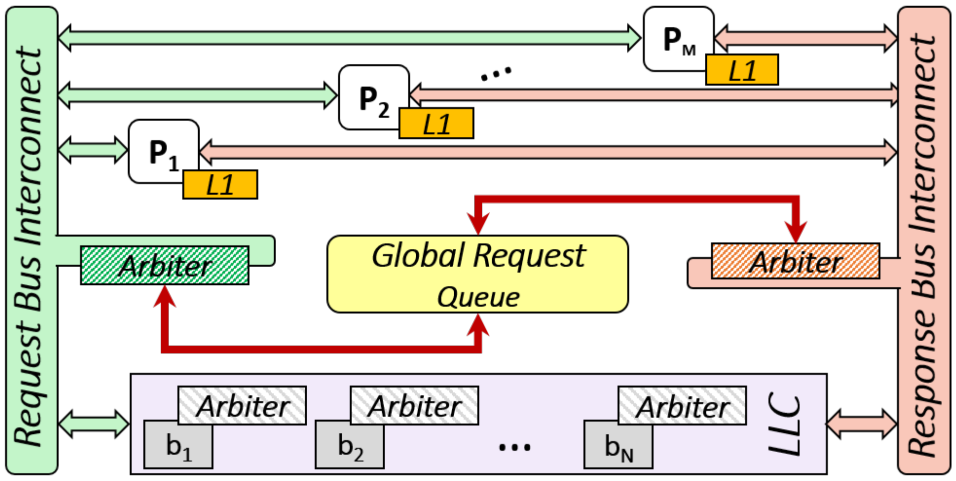


Figure 4.1: Architecture model.

4. DUEPCO operates alongside conventional coherence protocols without requiring any protocol modifications (addressing ***Problem ❷***).

5. Finally, we evaluate the proposed arbiter as well as DUEPCO against the state-of-the-art predictable coherency solution as well as a baseline COTS solution. Our evaluation shows that DUEPCO outperforms state-of-the-art predictable solution in terms of overall throughput by up to 6.4× and shows negligible slowdown compared to COTS solutions as low as 2% while providing comparable latency bounds to the best predictable mechanism.

4.1 System Model

In this section, we first detail the hardware architecture considered in this chapter along with the coherency assumptions. Then, we explain how requests generated by the cores are processed by the proposed hardware architecture and how the latency for each request is constructed.

4.1.1 Architecture and Coherency

An overview of the proposed hardware architectural model is delineated in Figure 4.1. We consider a multi-core system with M OoO requestors[1] including processing cores, $P_1, ..., P_i, ..., P_M$ where each requestor has exclusive access to a private cache. We assume that tasks running on the cores can share data among each other; hence, a coherency protocol must be employed in the system to allow coherent actions among cores and LLC. We assume data transfers amongst private caches could be either from the LLC banks or via the Cache-to-Cache (C2C) transfers which exhibit improved average-case performance. In this chapter, we adopt the MSI coherency protocol that includes three fundamental stable states as discussed in Chapter 2. Notice that, we do not modify the coherency protocol by any means, which simplifies the verification efforts compared to the approaches that alter protocols.

All cores have access to shared memory that we assume is an on-chip LLC. Instead of a unified interconnect commonly deployed in real-time architectures, we consider a split-transaction bus in which all communications between cores and LLC is done using two separate buses: 1) *request bus*, which is responsible for broadcasting the coherency messages; 2) *response bus*, which is dedicated interconnect to transfer the data responses from/to cores. These two buses operate in parallel to improve the performance of the system. The *request bus* and *response bus* take a certain amount of time to transfer message packet and data response, which we represent with t_{REQ} and t_{RESP}, respectively. In order to maximize the parallelism in the system, we propose to bankize the LLC in which multiple requests could be processed simultaneously. Therefore, we assume that the LLC consists of N independent banks, $b_1, ..., b_i, ..., b_N$ where each bank consumes a certain amount of time t_{BANK} to process writing data to (or retrieving data from) the cache data array inside each bank. In addition, LLC banks could be shared among all cores [42] or partially shared similar to [68]. In our model, we assume all banks are shared among cores. Similar to related work [39, 57], in this work, we only focus on L1-LLC traffic and do not model the extra delay in main memory due to LLC misses. The DRAM access latency can be computed using other approaches such as [115, 42, 44, 32] and such latency could be added to the WCL bounds derived in Section 4.3.

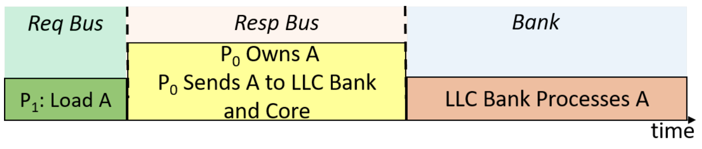
We assume that each cache entity (private and LLC) has its own set of interconnect buffers: `TxMsg`, `RxMsg`, `TxResp`, and `RxResp` to register the incoming/outgoing messages and data responses. `RxMsg` contains the incoming message packets from the request

[1]We use cores and requestors interchangeably throughout the chapter.

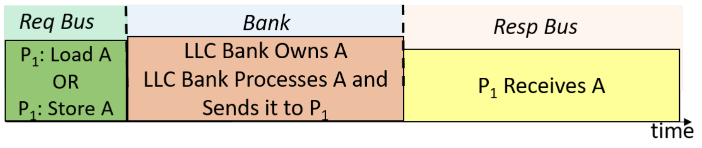
bus. We assume that every message will be decoded immediately in the private cache and each LLC bank even if the bank is busy writing/retrieving data from its data array. `TxMsg` contains the outgoing message packets from any core/bank that must snoop on the request bus. For instance, a core asking to modify a cache line and is not in possession of it must inform other cores by coherency message (GetM) and push it in its own `TxMsg` buffer to be propagated on the request bus. This allows other cores/LLC to be aware of this action. `RxResp` contains the data responses coming from the response bus. `RxResp` at each core includes data response that the bank provides or data response due to a C2C transfer. `RxResp` at LLC bank include the data response supplied by the cores in case of write-back. Notice that unlike `RxResp` buffers of each core, the data responses placed in LLC `RxResp` must be processed in the bank which takes t_{BANK} to process. Finally, `TxResp` includes the responses that need to be transferred on the response bus. The request bus, response bus, and LLC banks act as independent shared resources which conduct their own independent arbitration policies. In detail, the request bus arbiter is responsible to arbitrate the messages residing in `TxMsg` and the data responses inside `TxResp` buffers are arbitrated through the response bus arbiter. Similarly, each arbiter at LLC bank arbitrates the message/responses in `RxMsg` and `RxResp` buffers.

4.1.2 Request Processing and Order of Arbitration

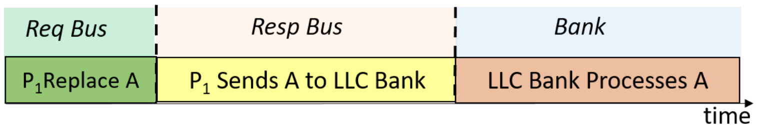
From the perspective of the coherency architecture, a requestor issues *requests* to the system based on the following activities in L1 cache: 1) load miss requests; 2) store miss requests, including stores to a cache line in **S** state and load miss requests; 3) replacement requests due to a write-back to shared memory or caused by an eviction. As mentioned earlier in this section, the proposed architecture applies the arbitration schemes at all different resources including request bus, response bus, and each bank in LLC. Requests can experience different sequences of services on the arbitration resources. In detail, we consider three different *types* − of requests, depending on the sequence of arbitration: 1) `REQ:RESP:BANK` meaning that it first needs to broadcast on the request bus, then the data response will be propagated on the response bus and finally, the data response should be processed at LLC bank; 2) `REQ:BANK:RESP` representing a category of requests in which they need to first broadcast on the request bus, then the shared bank must process and fetch the data response and finally this data response must be propagated over response bus; 3) `REQ:RESP`: the last category is related to the systems enabled with C2C transfers. In such a scenario, after broadcasting the message on the request bus, the response will be



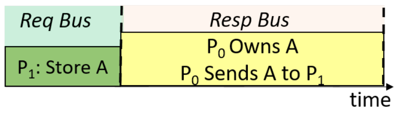
(a) P_1 executes load to A owned by P_0


Req Bus
Bank
Resp Bus
P$_1$: Load A OR P$_1$: Store A
LLC Bank Owns A
LLC Bank Processes A and Sends it to P$_1$
P$_1$ Receives A
time

(b) P_1 executes store/load to A owned by LLC bank.



(c) P_1 replace A to LLC bank.



(d) P_1 executes store to A owned by P_0.

Figure 4.2: The sequence of arbitration based on the request type.

supplied by the owner core on the response bus.

Figure 4.2 depicts all possible cases in which a request from cores can be processed based on its type and coherency status. Notice that, these cases are simplified and the details were omitted for readability purposes. Figure 4.2a represents a scenario where core P_1 aims to load cache line A; therefore, it first needs to broadcast its action by sending the required coherency message on the request bus. Since the owner of A is P_0 and the cache line A is in **M** coherency state; hence, according to **MSI** coherency protocol, P_0 must send A to both core P_1 and the LLC bank by pushing the response data into `RxResp` buffer of P_1 and the bank. Then, P_1 receives its data response; however, the data still needs to be processed inside the bank which will be done after t_{BANK}. Note that all of these actions are eligible to execute after their corresponding arbitration issue them the grant to access the resource. Hence, the sequence of arbitration for this request follows `REQ:RESP:BANK`.

In Figure 4.2b a load/store request from P_1 targets cache line A which is owned by the shared bank. Therefore, the bank is responsible to process the request, and then it can be returned to the core through the response bus. Hence, the sequence of arbitration for this request follows `REQ:BANK:RESP`. For the replacement request shown in Figure 4.2c, after broadcasting the message, the core needs to transfer the data to the LLC bank by pushing to the `RxResp` buffer of bank. Hence, the sequence of arbitration for this request follows `REQ:RESP:BANK`. Finally, Figure 4.2d shows a scenario where P_1 tries to store to cache line A while the line is owned by P_0. According to the **MSI** coherency protocol, the LLC bank is not required to acknowledge this action; therefore, sending the response from P_0 to P_1 suffices the store request and the sequence of arbitration for this request follows `REQ:RESP`.

As mentioned before, resources are independent and each resource needs to be arbitrated. However, to maintain the correctness of execution, the order of servicing the requests to the same cache line must respect the order in which the requests are issued on the request bus. We say that a request *depends* on the previous request to the same cache line issued on the request bus. Such dependencies can form chains of multiple requests. Due to dependencies, some requests might not move to the next resource even though their process is finished at the current resource. Specifically, we say that a request is *ready* on a resource if it can be considered for arbitration at that resource. If there is no dependency, a request becomes ready immediately when it finishes processing at its previous resource. On the other hand, if there exists a dependency, a request becomes ready at a resource when the previous request which caused the dependency finishes on that resource. Figure 4.3 shows an example where two requests follow the different sequences of resource

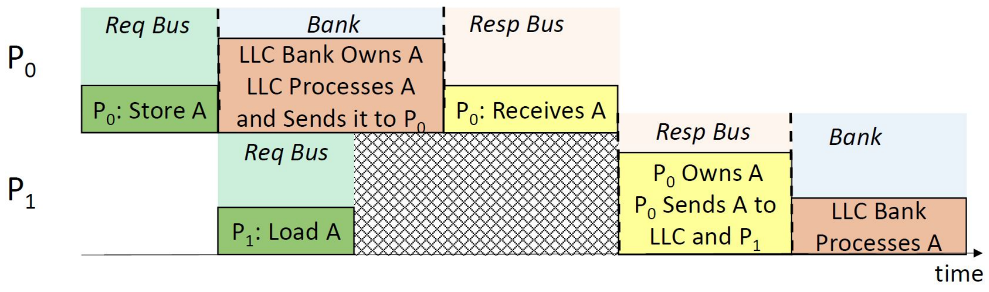


Figure 4.3: The lower priority request from P_1 *depends* on the higher priority request of P_0 to the same cache line A.

arbitrations such that request of P_0 follows the case elaborated in Figure 4.2d and request from P_1 follows the scenario shown in Figure 4.2a. Since both requests are to the same cache line A, request of P_1 depends on the request of P_0 and must only service after P_0 finishes at its response bus to maintain the consistency. Therefore, P_1 is not ready at response resource until P_0 finishes the response of its request.

4.1.3 Latency Model

Now, we are able to precisely define the request latency from the core perspective. As in Chapter 3, we index the requests of each core P_i as $r_{i,1}, ..., r_{i,j}, ...$. Each request has a type $\mathcal{T}$ according to its sequence. Since OoO cores might have multiple pending requests and the LLC contains many independent banks, they can serve multiple requests simultaneously. Since we are focusing on the interaction between L1 and LLC cache, we assume that the arrival time of a request is after the request has been processed by L1 cache. Therefore, the arrival time $t^a_{i,j}$ of a request $r_{i,j}$ corresponds to the time when it is queued in the `TxMsg` buffer. For our cache system, the finish time $t^f_{i,j}$ of $r_{i,j}$ is the clock cycle after which the last action in its sequence is completed. This includes writing to the shared bank if the type of sequence is `REQ:RESP:BANK` or receiving the response from the response bus if the type of sequence is `REQ:BANK:RESP` and `REQ:RESP`.

The same definition of outstanding and oldest request as in Chapter 3 apply. We also add the definition of *pending* for a request in this chapter to mean it is outstanding and already issued on the request bus.

4.1.4 Task Analysis

In Section 4.3, we will derive a bound on the processing latency for each of the three types of request defined in Section 4.1.2. The total access latency for a task can then be determined by summing the product of the number of requests of each type issued by the task by the WCL for that type [45].

We assume that a portion of accesses by the task targets data shared with other cores, while some accesses are to non-shared data. For each case, we need to retrieve the number of load miss requests, store miss requests, and the number of replacements from the task. For non-shared data, approaches based on either profiling or static analysis can be used to extract the number of requests. For shared data, to the best of our knowledge, no general method exists to determine which cache lines exist in the cache of the other cores at any point in time. A safe assumption can be adopted where every load request on shared data is considered a load miss, and every store request on shared data is considered a store miss [49]. However, if better assumptions can be made based on code analysis, our framework can take advantage of them by deriving different latency bounds for each type of request.

Note that based on Figure 4.2, for shared data, load misses can be from the type `REQ:RESP:BANK` or `REQ:BANK:RESP` shown in Figures 4.2a and 4.2b and store misses can be either `REQ:BANK:RESP` or `REQ:RESP`. For non-shared data, load and store misses can be only from the type of `REQ:BANK:RESP`. The replacements could only follow `REQ:RESP:BANK` as shown in Figure 4.2c. If we cannot determine the specific type of a request based on task analysis, we simply consider the largest latency among the types to which the request might belong.

4.2 Proposed Arbiter

This section describes the behavioral details of the proposed arbiter. The proposed arbiter considers the realistic hardware architecture introduced in Section 4.1 and maintains predictability by design while maximizing average-case performance. Based on the hardware architecture, there exist three distinct types of resources in the system. Formally, we capture the behavior of the proposed arbiter by a set of rules.

4.2.1 Rule 1: Global Round-Robin Ordering

In order to predictably manage interference among different cores, the arbiter maintains a unified Global Round-Robin (GRR) order of requestors across all resources. A requestor is removed from the GRR queue after the oldest request of that requestor completes at last resource, and it is inserted at the back of the queue either immediately when it has any other request or when its next request arrives. At any point in time, the *Global Request Queue* shown in Figure 4.1 contains all outstanding requests in the system as well as their state in terms of their next resource that they need to get processed on. In addition, a work-conserving approach is used at each resource to increase overall system performance.

Rule 1. The arbiter maintains a Global Round-Robin order of requestors across all resources.

Note that based on Rule 1, we can conceptually assign a relative GRR priority to each oldest request. Such priority is assigned when the request becomes oldest and never changes.

4.2.2 Rule 2: Bus Arbitration

We next explain the arbitration policy at each resource. Request bus arbiter arbitrates among the messages existing in the `TxMsg` of each core. The proposed arbiter deploys a two-level arbitration mechanism: 1) oldest message over non-oldest per core; 2) GRR order among the oldest messages. Obviously, if no oldest message exists, GRR over non-oldest messages will be applied. The same arbitration approach is true for the response bus and each LLC bank, but the arbitration will be applied to their corresponding interface buffers. Note that the arbitration at each LLC bank is completely independent of other banks in LLC.

Rule 2. The arbiter manages all the three resources, including request bus, response bus, and LLC banks, according to a two-level arbitration scheme: first, oldest request over non-oldest request, and second, following GRR from Rule 1.

4.2.3 Rule 3: Priority Inheritance

Rules 1 and 2 are sufficient in the absence of dependencies between requests. However, to correctly arbitrate in the presence of dependant requests, we further introduce a priority inheritance mechanism, whether a lower-priority request inherits the highest priority among following requests in the dependency chain; we call this the dynamic priority of the request. Rule 2 is then applied based on dynamic priorities.

Rule 3. A set of pending requests to the same cache line must be serviced in the same order in which they are issued on the request bus. For this reason, if there are multiple outstanding requests to the same cache line, a pending request is assigned a dynamic priority equal to the highest among its own priority and the priorities of all following requests to the same cache line. Ties in dynamic priorities are then broken based on dependency order (i.e., an earlier request in the dependency chain is considered higher priority than a following request).

4.2.4 Rule 4: Request Blocking

The proposed arbiter supports OoO execution, allowing processing cores to issue multiple pending requests. Based on Rule 3, it is clear that if the arbiter allows many non-oldest requests to the same cache line to be sent, then an oldest request could arrive and suffer priority inversion on all those non-oldest requests. Therefore, to limit the amount of priority inversion in the system, we set a parameter $k_{ceil} \geq 0$, that controls the possibility of sending non-oldest requests ahead of a possible oldest request to the same cache line.

Rule 4. The request bus arbiter blocks any non-oldest request $r_{i,j}$ once there are already other k_{ceil} pending non-oldest requests to the same cache line.

4.3 Latency Analysis

In this section, we detail the latency analysis for the proposed arbiter. Specifically, we first show how to compute an upper bound to the remaining latency (time to finish) for an oldest request *u*nder *a*nalysis r_{ua} at time t, based on the current state of the resource - following Chapter 3, we call this the *dynamic bound.* Then, we obtain the *static worst-case bound*, i.e. an upper bound to the processing latency of any request, by maximizing the dynamic bound over all possible states of the system.

4.3.1 Dynamic Latency Analysis

We will detail the analysis for a request under analysis targeting a generic bank b_k. Consider first a r_{ua} that does not depend on any other request. Depending on its type and its current state at time t, the r_{ua} will need to be serviced on one or more resources; for affinity with processor scheduling, we say that the r_{ua} must *execute* on those resources. Its remaining latency at time t can then be obtained as the sum of the latencies for each resource that r_{ua} executes upon; the latency for the first such resource is the interval between the current time t and when r_{ua} finishes executing on the resource; while the latency for each successive resource is the interval between r_{ua} becoming ready on that resource (which is the time when r_{ua} finishes executing on the previous resource, given that there is no dependency) and when r_{ua} finishes executing on that resource.

For a generic resource res with $res \in \{\texttt{REQ,BANK,RESP}\}$, where BANK denotes bank b_k, let $\mathcal{R}_{res}$ to denote the set of requests that have not yet started executing on res and either have higher dynamic priority than r_{ua} or are r_{ua} itself. Recall by arbitration Rule 2 that each resource arbiter follows a prioritized scheme; since a request cannot be stopped once it starts executing, request scheduling is non-preemptive. Hence, when the r_{ua} becomes ready on a resource res at some time $t' > t$, the currently executing request on that resource (if any) will first have to complete; in the worst case, such request is lower-priority and takes $t_{res} - 1$ clock cycles to complete, since it must have started before t'. Then, all requests in $\mathcal{R}_{res}$ must be processed, yielding a latency of $t_{res} - 1 + |\mathcal{R}_{res}| \cdot t_{res}$. For the resource where r_{ua} needs to execute at time t, which we denote as $\overline{res}$, a tighter bound can be obtained. Specifically, we use $c_{\overline{res}} < t_{res}$ to denote the *timer* for res, that is, the number of clock cycles required to complete the currently executing request; if no request is in progress on res at time t, we have $c_{\overline{res}} = 0$. Then, if the currently executing request is r_{ua}, its latency is by definition $c_{\overline{res}}$; otherwise, its latency is $c_{\overline{res}} + |\mathcal{R}_{res}| \cdot t_{res}$. To obtain an expression for the overall remaining latency, let us use $\mathcal{S}_{ua}$ to denote the set of resources where r_{ua} has not yet started executing. Then by summing the latencies for the first resource and subsequent resources we obtain:

$$c_{\overline{res}} + \sum_{res \in \mathcal{S}_{ua} \setminus \overline{res}} (t_{res-1}) + \sum_{res \in \mathcal{S}_{ua}} |\mathcal{R}_{res}| \cdot t_{res} \tag{4.1}$$

We next discuss the case where r_{ua} depends on one or more previous requests to the same cache line; note that all such requests must also target bank b_k. As discussed in Section 4.1, such chain of dependent requests creates precedence constraints: specifically, a request in the chain cannot become ready on a resource before the previous request in the chain (if any) has completed execution on that same resource. Therefore, to bound the remaining latency for the r_{ua}, we now need to sum the latencies over all resources accessed over the chain of precedence constraints. As an example, assume that the r_{ua} is of type `REQ:BANK:RESP`, that it depends on a request r' of type `REQ:RESP:BANK`, and that both requests have not completed execution on `REQ` yet. If the r_{ua} finishes execution on `REQ` before r' finishes on `BANK`, then the chain of precedence constraints is `REQ(`r'`):RESP(`r'`):BANK(`r'`):BANK(`r_{ua}`):RESP(`r_{ua}`)`. Otherwise, it is `REQ(`r'`):REQ(`r_{ua}`):BANK(`r_{ua}`):RESP(`r_{ua}`)`. Since in general we do not know which chain leads to the largest latency, we will overapproximate the chain of precedence constraints as `REQ(`r'`):REQ(`r_{ua}`):RESP(`r'`):BANK(`r'`):BANK(`r_{ua}`):RESP(`r_{ua}`)`. Similarly, if r' was of type `REQ:BANK:RESP` instead, the overapproximated chain would be `REQ(`r'`):REQ(`r_{ua}`):BANK(`r'`):BANK(`r_{ua}`):RESP(`r'`):RESP(`r_{ua}`)`.

Next, consider how to compute the latency for each resource in the chain of precedence constraints. First, consider the case where a resource res is accessed sequentially by two or more requests (e.g., `BANK(`r'`):BANK(`r_{ua}`)` in the first example where r' and r_{ua} have different types). Note that for a request to be delayed by a previous request in the sequence on res, the request must become ready on res when the previous one finishes. Hence, no lower-priority request can be executed on res in-between the execution of requests in the sequence. Therefore, the latency bound is still equal to $t_{res} - 1 + |\mathcal{R}_{res}| \cdot t_{res}$. Second, consider the case where a resource res is not accessed sequentially (e.g., `RESP(`r'`)` later followed by `RESP(`r_{ua}`)`). Here, every high-priority request in $\mathcal{R}_{res}$ still only executes once on res, hence it can only delay one of the requests in the chain. However, every time one of the requests in the chain becomes ready on res, it can suffer blocking by one lower-priority request. Hence, for this example, the latency bound on `RESP` is $2 \cdot (t_{RESP} - 1) + |\mathcal{R}_{RESP}| \cdot t_{RESP}$.

To generalize the latency expression over a chain of precedence constraints, let us redefine $\mathcal{S}_{ua}$ as the set of resources where at least one request in the chain has not yet started executing; and let $\overline{res}$ be the first resource in the chain at time t (such that a request still needs to finish execution on that resource). Furthermore, let us define K_{BANK} (K_{RESP}) to be the number of times that the `BANK` (respectively, `RESP`) resource is encountered in the chain of precedence constraints while being preceded by another resource. Then, the remaining latency for r_{ua} at time t can be upper bounded as:

$$c_{\overline{res}} + \sum_{res \in \mathcal{S}_{ua}} |\mathcal{R}_{res}| \cdot t_{res} + K_{BANK} \cdot (t_{BANK} - 1) + K_{RESP} \cdot (t_{RESP} - 1). \tag{4.2}$$

Note that in Equation 4.2, we do not need to consider a term K_{REQ} because all requests must first execute on `REQ`; hence, there can only be a single sequence of `REQ` accesses at the beginning of the chain (unless all requests have already finished executing on `REQ`, in which case we do not need to consider `REQ` latency). Furthermore, if r_{ua} has not yet started executing on `REQ`, then the chain of dependencies must be created including all oldest requests to the same cache line with priority greater than r_{ua}, even if those requests have not been executed on `REQ` and are thus not pending yet; this is because following the GRR order, such requests will execute on `REQ` before r_{ua} and thus r_{ua} will become dependent on them.

Example: consider a chain of dependency with requests r', r'' and r_{ua}. r' and r'' are of type `REQ:RESP:BANK`, while r_{ua} is of type `REQ:BANK:RESP`. Assume that at time t, all three requests have finished executing on `REQ`, and that r' is executing on `RESP`. Then, the chain is `RESP(`r'`):RESP(`r''`):BANK(`r'`):BANK(`r''`):BANK(`r_{ua}`)` `:RESP(`r_{ua}`)`; we have $\mathcal{S}_{ua} = \{$`RESP,BANK`$\}$, $\overline{res} =$ `RESP`, $K_{BANK} = K_{RESP} = 1$ (specifically, for K_{BANK} and K_{RESP} we need to count `BANK(`r'`)` and `RESP(`r_{ua}`)` as they are the only resources preceded by a different resource). Note that $\mathcal{R}_{BANK}$ includes r', r'' and r_{ua}, but $\mathcal{R}_{RESP}$ only includes r'' and r_{ua}. The latency bound is:

$$c_{RESP} + |\mathcal{R}_{RESP}| \cdot t_{RESP} + |\mathcal{R}_{BANK}| \cdot t_{BANK} + t_{BANK} - 1 + t_{RESP} - 1, \tag{4.3}$$

where c_{RESP} is the remaining execution time for r' on `RESP`.

The final note is related to the sets $\mathcal{R}_{REQ}, \mathcal{R}_{BANK}, \mathcal{R}_{RESP}$. For the latency bound to be applicable, the membership of these sets should be evaluated at time t. Hence, for the bound to hold, we need to prove that the cardinality of each set cannot increase after t.

Lemma 6. *Assume that r_{ua} is oldest at time t and consider any other request $r_{i,j}$. If $r_{i,j}$ has lower dynamic priority than r_{ua} at t, or has not arrived in the system yet, then its dynamic priority cannot be higher than r_{ua} at any point after t.*

Proof. We prove the lemma in two parts. First, we show that (1) the dynamic priority of r_{ua} cannot decrease over time; this implies that a request $r_{i,j}$ with dynamic priority in between the GRR and dynamic priority of r_{ua} at t cannot become higher priority than r_{ua} simply because the dynamic priority of r_{ua} drops. Therefore, $r_{i,j}$ would have to acquire a new dynamic priority higher than the one of r_{ua} at t; but we next show that this is impossible. Note that because of the GRR ordering, a request $r_{p,q}$ that becomes oldest after r_{ua} will have a lower GRR priority than r_{ua}. This also means that $r_{i,j}$ cannot acquire a dynamic priority higher than r_{ua} by inheriting the priority of such $r_{p,q}$. Finally, to conclude the lemma we show that (2) no request $r_{i,j}$ can inherit after t the priority of a request $r_{p,q}$, where $r_{p,q}$ is either r_{ua} or a request with GRR priority higher than r_{ua}.

Part (1): since the GRR priority never changes, the dynamic priority could only decrease if r_{ua} is inheriting the priority of a following, higher priority request $r_{p,q}$ in the dependency chain at t, and then such request finishes before r_{ua}. However, this is impossible because dependencies force requests to finish according to the dependency order.

Part (2): since $r_{p,q}$ is either r_{ua} or has higher GRR priority, it follows that $r_{p,q}$ must already be oldest at t. If $r_{p,q}$ has already started executing on `REQ`, then it cannot become dependent on any new request after t. If $r_{p,q}$ has not started executing on `REQ`, then the `REQ` arbiter will favor executing requests in priority order, hence no non-oldest request or oldest request with GRR priority lower than $r_{p,q}$ can start executing on `REQ` before $r_{p,q}$. In both cases, it follows that if $r_{p,q}$ is not dependent on $r_{i,j}$ at t, then it cannot become dependent on $r_{i,j}$ after t and thus $r_{i,j}$ cannot inherit the priority of $r_{p,q}$.

□

4.3.2 Static Analysis

The static worst-case latency Δ is the maximum remaining latency of any request at the time t when it becomes oldest. We compute it by maximizing Equation 4.2 over all possible values of the parameters. Note that the equation is maximized when the r_{ua} has not yet started executing on `REQ`. Hence, we have $\overline{res} = \texttt{REQ}$ and in the worst-case $c_{REQ} = t_{REQ} - 1$; if $k_{ceil} = 0$, then $\mathcal{S}_{ua} = \{\texttt{REQ,RESP}\}$ for a request of type `REQ:RESP` and $\mathcal{S}_{ua} = \{\texttt{REQ,BANK,RESP}\}$ otherwise; while if $k_{ceil} > 0$, in the worst-case a previous request in the dependency chain can use all three resources, hence we have $\mathcal{S}_{ua} = \{\texttt{REQ,BANK,RESP}\}$:

$$\mathcal{S}_{ua}(\mathcal{T}, k_{ceil}) = \begin{cases} \{\texttt{REQ,RESP}\} & \text{if } k_{ceil} = 0 \land \mathcal{T} = \texttt{REQ:RESP} \\ \{\texttt{REQ,BANK,RESP}\} & \text{otherwise} \end{cases} \tag{4.4}$$

We next consider K_{BANK} and K_{RESP}. The worst-case scenario is to alternate the type of requests in the dependency chain between REQ:BANK:RESP and REQ:BANK:RESP. Based on the type $\mathcal{T}$ of the request under analysis, this yields:

$$\begin{aligned}
K_{RESP}(\texttt{REQ:BANK:RESP}) &= 1 + \lceil k_{ceil}/2 \rceil, && (4.5)\\
K_{BANK}(\texttt{REQ:BANK:RESP}) &= 1 + \lfloor k_{ceil}/2 \rfloor, && (4.6)\\
K_{RESP}(\texttt{REQ:RESP:BANK}) &= 1 + \lfloor k_{ceil}/2 \rfloor, && (4.7)\\
K_{BANK}(\texttt{REQ:RESP:BANK}) &= 1 + \lceil k_{ceil}/2 \rceil, && (4.8)\\
K_{RESP}(\texttt{REQ:RESP}) &= 1 + \lfloor k_{ceil}/2 \rfloor, && (4.9)\\
K_{BANK}(\texttt{REQ:RESP}) &= \lceil k_{ceil}/2 \rceil. && (4.10)
\end{aligned}$$

Finally, we consider the sets $\mathcal{R}_{REQ}, \mathcal{R}_{BANK}, \mathcal{R}_{RESP}$. Since there are M requestors in the system, the number of requests with GRR priority higher than or equal to the request under analysis is at most M. Furthermore, because arbitration Rule 4 limits the number of non-oldest requests to the same cache line to k_{ceil}, it follows that at most k_{ceil} non-oldest requests can inherit the priority of each of the M oldest requests. Therefore, counting the request under analysis itself, in the worst-case we have $|\mathcal{R}_{BANK}| = |\mathcal{R}_{RESP}| = (k_{ceil}+1) \cdot M$. The latency on REQ can be more accurately bounded. Based on Rule 3, only requests that are pending, i.e. have already finished executing on the REQ bus, can inherit the priority of another request. Hence, in the worst-case we have $|\mathcal{R}_{REQ}| = M$.

Substituting the computed parameters into Equation 4.2 yields:

$$\begin{aligned}
&\Delta(\mathcal{T}, k_{ceil}) = t_{REQ} - 1 + M \cdot t_{REQ} + (k_{ceil}+1) \cdot M \cdot t_{RESP}\\
&+ K_{BANK}(\mathcal{T}) \cdot (t_{BANK} - 1) + K_{RESP}(\mathcal{T}) \cdot (t_{RESP} - 1)\\
&+ \begin{cases} 0 & \text{if } k_{ceil} = 0 \wedge \mathcal{T} = \texttt{REQ:RESP} \\ (k_{ceil}+1) \cdot M \cdot t_{BANK} & \text{otherwise} \end{cases}
\end{aligned} \qquad (4.11)$$

4.4 DUEPCO: Duetto Application for Coherency

In this section, we discuss how the Duetto reference model is applied to our discussed cache system to form the DUEPCO architecture. As in Chapter 3, the system designer associates a relative deadline $D_i(\mathcal{T})$ to each requestor and type of request; as long as $D_i(\mathcal{T}) \leq \Delta(\mathcal{T}, k_{ceil})$, Duetto then guarantees that all request deadlines will be met.

Note that the simple resource considered in Chapter 3 only requires a single command to satisfy/finish a request; in contrast, in our cache design, a request must be serviced on either two or three resources depending on its type. Hence, a request requires multiple commands to complete, and a more complex state machine is required to track the state of each cache line through the coherency protocol. In addition, we extend the reference model in two ways to introduce DUEPCO. First of all, it is important to notice that for the Duetto deadline guarantee to hold, the static worst-case latency must be computed assuming any valid state of the resource at the time t when the request under analysis becomes oldest; this is because the HPA might be selected at any time before t. However, when we computed the static latency in Section 4.3.2, we bounded the cardinality of sets $\mathcal{R}_{BANK}$ and $\mathcal{R}_{RESP}$ assuming that arbitration Rule 4 always applies, as this ensures that no more than k_{ceil} non-oldest requests can inherit the priority of an oldest request. Unfortunately, the HPA does not need to satisfy such rule, and can instead execute any number of requests to the same cache line on the `REQ` bus before an oldest request to that line arrives and its latency is considered by the WCLator; at which point it is too late to switch to the RTA.

To address this issue, we make a conceptual change to the model of the resource. Specifically, we declare that all states where there are more than k_{ceil} pending non-oldest requests to the same cache line are invalid to avoid too many priority inversions. This ensures that the derived static bound is correct, but does not solve the underline problem as now the HPA might be issuing invalid commands; therefore, a logic must be incorporated to block invalid commands to issue. Chapter 3 suggests that when the HPA cannot be guaranteed to work correctly, a checker module can be added to check the validity of the commands issued by the HPA. Therefore, DUEPCO adds an additional checker component that works as follows: every clock cycle, the checker receives from the RTA information on the number of pending requests per cache line, which the RTA maintains to enforce Rule 4. If $c_{REQ} = 0$ and the global request queue contains at least one non-oldest request that must execute on the `REQ` bus and targets a cache line for which there are k_{ceil} pending non-oldest requests, the HPA might issue such request on `REQ` and reach an invalid resource state. Hence, in this case the checker overrides the WCLator to forcibly select the RTA. While this approach solves the unbounded priority inversion problem, it has a downside: for low values of k_{ceil} and/or heavy data sharing among requestors, the checker might be forced to continuously select the RTA, resulting in performance loss. We explore this behavior in more details in the evaluation Section 4.5.

The reference model in Chapter 3 assumes that the type of a request is known when the request becomes oldest. However, in our system, the sequence of resources accessed by a request, and thus its type, is only known after the request is executed on the `REQ` bus and the owner of the corresponding cache line is determined. For this reason, before an oldest request finishes executing on `REQ`, the WCLator must use the smallest among all deadlines for the possible types for the request. Once the request type is known, the WCLator switches to using the deadline for that type.

4.4.1 WCLator Design

We designed the WCLator following the methodology outlined in Chapter 3. For each oldest request and given the state of the resource and global request queue, we first enumerate all commands that the HPA could issue in this clock cycle. For each command, we then use the dynamic analysis (possibly with modified value of the parameters) in Section 4.3 to compute the remaining latency for the request. Since the WCLator is a hardware component, all such cases can be estimated in parallel. The WCLator then compares the largest computed latency against the deadline for the request to determine whether the HPA can be selected.

Consider an oldest request r_{ua} at time t. To illustrate the behavior of the WCLator, we enumerate the cases assuming that $\overline{res} = \texttt{REQ}$ (the cases for $\overline{res} = \texttt{BANK}$ and $\overline{res} = \texttt{RESP}$ are similar but easier, since the dependency chain for r_{ua} cannot be affected):

1. If $c_{REQ} > 0$, then no command can be issued on `REQ` in this clock cycle, and no estimation is required. Note that if $c_{BANK} = 0$ or $c_{RESP} = 0$, the HPA could start executing a request on `BANK` or `RESP` in this clock cycle; however, because Equation 4.2 always assumes the worst case where the maximum blocking time is suffered on successive resources, it follows that the bound is still safe no matter the command issued by HPA on `BANK` and/or `RESP`. Therefore, for the remaining cases, we assume $c_{REQ} = 0$ and consider the command issued on `REQ`.

2. No command: the HPA might be non-work conserving and decide to issue no command in the current cycle. In this case, the bound is equal to Equation 4.2 plus one, to account for the wasted clock cycle.

3. r_{ua}: since r_{ua} will start executing, we would need to apply Equation 4.2 after removing it from $\mathcal{R}_{REQ}$, but setting $c_{REQ} = t_{REQ}$ to account for the r_{ua} execution. In addition,

if there are higher-priority requests to the same cache line as r_{ua} which have not yet executed on `REQ`, we would need to remove such requests from $\mathcal{R}_{BANK}, \mathcal{R}_{RESP}$ and adjust K_{BANK}, K_{RESP}, since r_{ua} now executes before them. Note that the obtained bound will always be lower than case 2); therefore, in practice the WCLator does not need to consider this case.

4. A lower-priority request $r_{i,j}$, which does not inherit a higher priority than r_{ua} after becoming pending: we use Equation 4.2 with $c_{REQ} = t_{REQ}$.

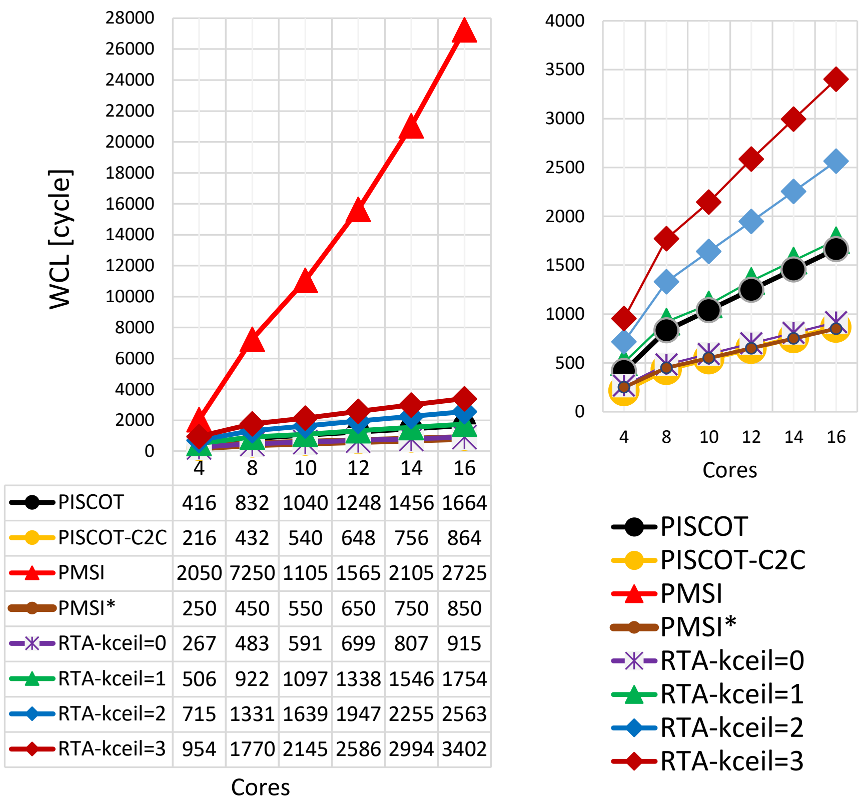
5. A lower-priority request $r_{i,j}$ that inherits a higher priority than r_{ua}: in addition to the previous case, we need to include the request in $\mathcal{R}_{BANK}, \mathcal{R}_{RESP}$. Furthermore, if $r_{i,j}$ targets the same cache line as r_{ua}, K_{BANK}, K_{RESP} must be adjusted.

6. A higher-priority request $r_{p,q}$ to a different cache line than r_{ua}: we use Equation 4.2 with no change to parameters. Since this bound is always lower than 2), again we do not need to consider it.

7. A higher-priority request $r_{p,q}$ to the same cache line of r_{ua}: before applying Equation 4.2, parameters K_{BANK}, K_{RESP} need to be adjusted if $r_{p,q}$ is executed before another higher-priority request to the same cache line.

4.5 Evaluation Results

We employed an open-source simulation framework provided by [49] to evaluate the performance of the proposed mechanisms and compare them with other solutions. We emulate a system with quad- and octa-core system clocked at 2.5GHz with out-of-order pipelines, 8 KB direct-mapped L1 per-core private cache, and a 4 MB 8-ways set-associative L2 shared cache consisting of multiple separated banks. The cores are OoO and can issue up to 10 memory requests in parallel. Both L1 and LLC have a cache line size of 64 bytes. Each core/LLC bank is equipped with a dedicated cache controller that implements the MSI coherence state machine. Since we are considering multi-core systems with strict timing guarantees, we name the proposed arbiter in Section 4.2, RTA and compared it against state-of-the-art approaches including PMSI [39], PMSI* [56] and PISCOT [49] which also provide analytical WCL bounds and present the best average-case performance. PMSI employs unified bus architecture and provides relative high-performance gains compared to other approaches such as shared data-aware scheduling and private cache bypassing



Cores	4	8	10	12	14	16
● PISCOT	416	832	1040	1248	1456	1664
● PISCOT-C2C	216	432	540	648	756	864
▲ PMSI	2050	7250	1105	1565	2105	2725
● PMSI*	250	450	550	650	750	850
✳ RTA-kceil=0	267	483	591	699	807	915
▲ RTA-kceil=1	506	922	1097	1338	1546	1754
◆ RTA-kceil=2	715	1331	1639	1947	2255	2563
◆ RTA-kceil=3	954	1770	2145	2586	2994	3402

Figure 4.4: Per-request worst-case latency.

through deploying cache coherence modifications and accessing the shared data. However, its WCL is quadratic in the number of cores in the system. PMSI* follows a systematic approach that achieves the same static WCL as bypassing the shared cache and provides a tighter WCL bound compared to PMSI. However, both of these techniques rely on many coherency modifications and expose performance loss compared to other approaches. On the other hand, PISCOT decouples the request and response bus and leverages the split-transaction interconnect to achieve a tighter WCL compared to PMSI and considerable performance gains.

Request bus latency is configured to 4 cycles ($t_{REQ} = 4$). The response bus latency in PISCOT is comparable to the TDM slot size in PMSI as well as PMSI* and we set them to 50 cycles in our evaluation similar to [49]. However, for RTA, the latency of all resources is configurable. Throughout this section, unless otherwise specified, we configure RTA with $t_{RESP} = 10$, 8 banks that consume $t_{BANK} = 40$ to process requests and parameter $k_{ceil} = 1$.

Similar to existing works [57, 39, 49], we assume that accesses that hit in the L1 cache take a single clock cycle and LLC is a perfect cache to avoid extra delay from accessing the off-chip memory subsystem.

We craft three sets of synthetic benchmarks (`Synth 1`, `Synth 2`, `Synth 3`) with different characteristics. All contain mixed read and write requests to the LLC and we engineered the requests' addresses such that all requests miss in the L1 cache; hence, stress on the bus and the shared cache banks will be maximized. There is no data-sharing among the cores in `Synth 1` while `Synth 2` and `Synth 3` exhibit 10% and 20% data-sharing respectively. In all benchmarks, the foreground core represents a high load core that bursts requests to bus/LLC, and the background cores are accessing the shared bus/LLC less frequently. Interleaving across the banks is handled using address bits themselves such that a core could access all banks as much as possible. In detail, we use bit 6^{th} (bits zero to 5^{th} are for the cache line offset) towards the MSB in the address bits of the request to determine which LLC bank it needs to be processed in.

4.5.1 Per-Request Worst-Case Latency

Figure 4.4 shows the static WCL bounds for requests generated by the cores and misses in L1 caches (see Section 4.3) from `REQ:RESP:BANK` type which represents the largest static WCL among three types. We compare PMSI, PMSI*, PISCOT and PISCOT-C2C (with core to core transfers), and the proposed RTA mechanism with different values of parameter k_{ceil}. From this experiment, we can make the following observations: 1) PMSI shows a significantly higher latency bound compared to the other approaches, and the latency bound increases quadratically with scaling the number of cores. The significant added latency is due to the coherence interference on the shared data. PMSI* on the other hand presents tight static WCL bound but at the cost of performance degradation [49, 56]; 2) PISCOT shows looser bound compared to both PISCOT-C2C but similar to RTA since core to core transfers enable the arbiters to bypass the LLC when an owner core must respond to other cores; 3) RTA with $k_{ceil} = 1$ shows up to 1.18× looser bound compared to PISCOT-C2C but significantly tighter bound compared to PMSI. Notice that this extra amount in latency bound is due to the scheduling decisions that are made in RTA which allow one non-oldest request to process in LLC banks. This gives the system a significant advantage in terms of average performance as we will show in the next sections. It is worthwhile to stress the existing trade-off between RTA with different values of k_{ceil} and PISCOT-C2C. RTA with $k_{ceil} = 0$ represents a configuration in which no non-oldest request

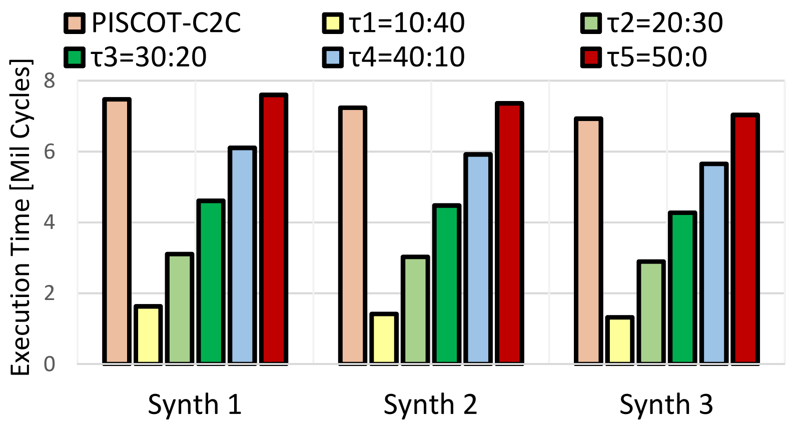
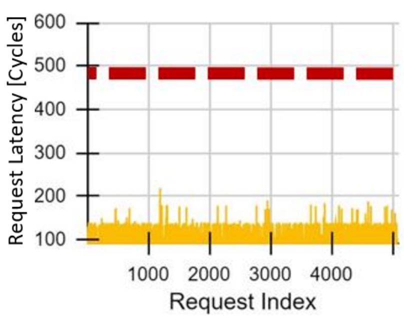


Figure 4.5: Sensitivity test for RTA against PISCOT-C2C.

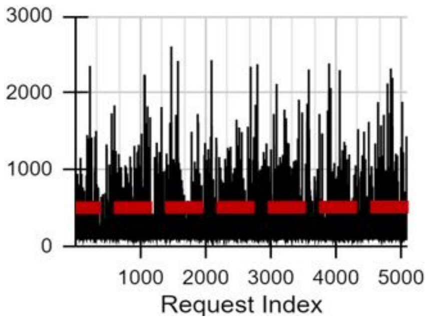
is allowed to process in the shared banks. This improves the WCL bound such that it becomes tighter and very similar to PISCOT-C2C. However, DUEPCO does not work with this configuration of RTA ($k_{ceil} = 0$) since the checker module is forced to select the RTA if there is any non-oldest request needing to be serviced on the request bus.

4.5.2 Sensitivity Test

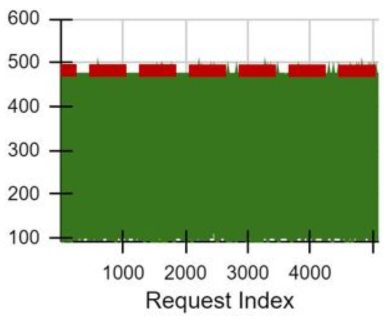
The underlying architecture proposed in Section 4.1 is fully configurable to resemble the conventional high-performance bus/LLC designs. In this section, we conduct a sensitivity test on the RTA to justify the most efficient (and the worst) design that is aligned with commercial architectures and compare it against PISCOT-C2C. We configured a quad-core system with $t_{REQ} = 4$ and then gradually varied t_{BANK} and t_{RESP} latencies. In order to run a fair comparison, the parameters are determined such that $t_{RESP} + t_{BANK} = 50$, the response bus latency for PISCOT-C2C. Assuming $\tau = t_{RESP} : t_{BANK}$ represents a configuration of RTA in which the latency of shared banks in LLC is t_{BANK} and the latency of response bus equals t_{RESP}, Figure 4.5 shows the execution time of the foreground core running each of the three synthetic benchmarks. As discussed, RTA increases the parallelism through bankized LLC. Therefore, as we increase t_{BANK} in LLC and coincidentally decrease t_{RESP}, we observe that the system performance improves by finishing the task under analysis faster. In other words, by reducing the response bus latency, a significant amount of arbitration stress will be transferred to the banks rather than the response bus; hence, the system's overall performance increases by allowing more transactions to be



(a) Observed request latency under RTA



(b) Observed request latency under HPA.



(c) Observed request latency under DUEPCO

Figure 4.6: Observed latencies under different arbitration schemes.

serviced simultaneously. In detail, the core under analysis in RTA, τ_1 running `Synth 1` outperforms PISCOT-C2C by 4.58× in terms of overall throughput of the system. Note that in τ_5 where there is no parallelism in RTA, we observe a negligible performance loss compared to PISCOT-C2c (maximum 1% in overall throughput) since response bus arbiter in PISCOT-C2C is FCFS while RTA employs a fair round-robin mechanism through GRR. Notice that τ_5 conceptually represents a configuration similar to PISCOT-C2C with only one bank but allowing multiple requests of the same requestor. Therefore, as it is clear from the figure, relaxing PISCOT-C2C to issue multiple outstanding requests with one bank (similar to tau_5) does not deliver any performance gain.

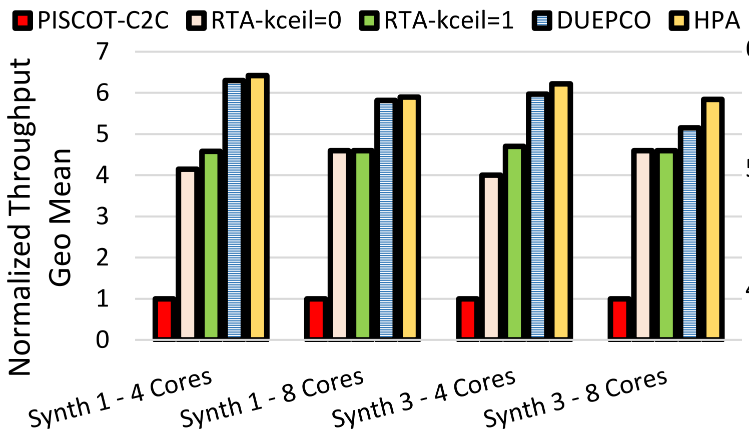
Going forward, we chose τ_1 as it resembles the configuration with a higher level of parallelism resembling a more realistic architecture.

4.5.3 Observed Request Latency

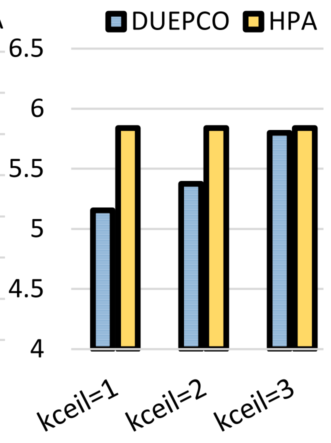
Figures 4.6a, 4.6b, and 4.6c delineate the observed latency suffered by oldest miss requests from `REQ:BANK:RESP` type generated by a quad-core system under RTA, HPA, and DUEPCO. We show request latencies greater than 80 cycles for better visibility and run the experiment with `Synth 3` benchmark (other benchmarks/request types show similar behavior). The RTA latency bound for this setup is 476 cycles based on the derived WCL analysis in Section 4.3 which is shown as a red bar in the figures. In HPA, we observe large latency spikes throughout the execution up to 3420 cycles since HPA favors requests from the cores that generate the highest number of requests, are faster, and target the banks that are idle which can starve (theoretically) or delay for a long time (practically) requests targeting busy banks. Figure 4.6a shows that RTA respects the latency bound for all requests from every core and the latencies are always below the WCL bound. However, there is a gap between the latencies and the static WCL bound since static analysis conducted in Section 4.3 must assume that the oldest requests of all cores access the same bank at the same time in addition to the non-oldest requests, which is unlikely in practice. Finally, in DUEPCO, we used the static WCL bound as the deadline for each oldest request. Figure 4.6c shows that DUEPCO stretches the latency of requests towards the latency bound and allows the system to continue selecting the HPA as long as possible.

4.5.4 Average Performance: Throughput

To measure the average performance of DUEPCO, we use the total throughput of the system. As before, we associate the static WCL bound as the deadline to the oldest requests in DUEPCO. Figure 4.7a shows the geometric mean of throughput across all cores for RTA HPA and DUEPCO normalized to the overall throughput of PISCOT-C2C. The figure represents the results for four different setups: 1) a quad-core system running `Synth 1`; 2) a quad-core system running `Synth 3`; 3) an octa-core system running `Synth 1`; 4) an octa-core system running `Synth 1`. We make the following observations: 1) RTA, HPA, and DUEPCO outperform the single-bank architecture approach deployed in PISCOT-C2C significantly, by up to 6.4×; 2) DUEPCO shows very small slowdown compared to HPA in `synth 1` and `synth 3 - 4 core` (at most 2%); 3) in an octa-core system and `synth 3` benchmark, we observe a slowdown of 11%. Following the discussion in Section 4.4, since DUEPCO employs RTA with $k_{ceil} = 1$, it has to exclude the invalid states from the HPA by switching to RTA$\dot{R}$ecall that `Synth 3` benchmark expose 20% data-sharing among the cores, and this leads to the case that multiple cores compete to access the same cache line in a particular bank. Therefore, DUEPCO selects the RTA regardless of the WCLator estimation according to the checker logic. However, by increasing the number of allowed requests to the same cache line (k_{ceil}), we expect that DUEPCO selects the HPA more often. As shown in Figure 4.7b, DUEPCO that employs RTA with $k_{ceil} = 3$ exhibits only 1% slowdown compared to HPA. Notice that relaxing the parameter k_{ceil} forces us to use a higher value for the static WCL bound for each oldest request as shown in Figure 4.4. It is also worth noting that RTA with $k_{ceil} = 0$ is a similar case to PISCOT-C2C with multiple bank support but with only one request per requestor. As it is clear from the figure, the performance gain is still lower that the RTA with $k_{ceil} = 1$ and DUEPCO.



(a)



(b)

Figure 4.7: Total throughput of the system.

4.6 Summary

Employing shared memory in multi-core platforms improves programmer productivity and degrades the obstacle to using such platforms in real-time systems. Hardware cache coherence can accommodate such shared memory and extend the advantages of on-chip caching to all system memory. However, extending hardware cache coherence throughout traditional schemes such as coherency protocol modifications to provide predictability hurts the performance of the system. In this work, we demonstrate that by employing the COTS interconnect architecture along with proposing to bankize the on-chip cache, DUEPCO is able to pair a clever global arbitration mechanism with Duetto to significantly improve the performance of the system while providing predictability. Notice that while we propose DUEPCO with simple buses, potentially the same arbitration scheme could be added to other bus architectures such as AXI in ARM platforms. However, the fundamental constraint to consider is that the arbiter must have exclusive visibility into the queues of each requestor.

Chapter 5

DuoMC: Applying Duetto to DRAM with Shared Banks

In this chapter and inspired by the Duetto reference model introduced in Chapter 3, we propose DuoMC: a 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 [26, 47, 117, 74] and it is one of the critical bottlenecks both from latency as well as performance [67, 80, 36] perspectives. Unlike most existing real-time MCs, DuoMC enables the utilization of both private and shared DRAM banks among cores to facilitate communication among tasks. In summary, we make the following contributions in this chapter.

1. DuoMC adopts the Duetto reference model, such that it can be modularly integrated into existing COTS MCs with minimal hardware modifications in the HPA and without requiring detailed information on the internal behavior of the already implemented HPA in the COTS platform. Unlike the simplified abstract SRAM resource in Chapter 3, 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 [106]. As a result, DuoMC extends and generalizes the conceptual model introduced in Chapter 3 to be applicable to more realistic shared resources existing in modern COTS SoCs. This generalization is discussed in detail in Section 5.1. Specifically, we show how to distinguish between completion (end of service) and finish (end of latency) time of a request (Section 5.1).

2. We propose a novel real-time MC scheduler, RTA (Section 5.2) in which real-time guarantees are achieved by monitoring the latencies incurred by DRAM requests in the system and switching from a COTS HPA to RTA only in the rare cases when these guarantees are at the risk of being violated.

3. Unlike most of the existing predictable MCs [22, 23, 33, 74, 84], 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 5.4.2). This is one of the key contributions of this chapter since most industrial embedded domains such as automotive and avionics [35] require communication among different tasks/processing components through shared data [17].

4. We conduct a detailed timing analysis of DuoMC, which provides guaranteed bounds on the WCL suffered by any request to the DRAM (Section 5.4).

5. We provide a detailed evaluation of DuoMC by implementing it in MacSim [60], a multi-core full-system, cycle-accurate simulator. Our results show that DuoMC suffers only 8% performance degradation across EEMBC 1.1 auto benchmark suite [91] and IsolBench [110] benchmarks compared to a high-performance memory controller (Section 5.7) while providing comparable WCL to state-of-the-art predictable MCs.

5.1 DuoMC: The Proposed Solution

In this section, we discuss how to apply the Duetto reference model to the DRAM resource to create DuoMC. Here we focus on adaptations and changes to the reference model, while Section 5.5 discusses some of the lower-level details that are specific to the implementation. As in Chapter 3, we consider a system with M requestors: $P_1, \ldots, P_M$ and N DRAM banks $b_1, \ldots, b_N$, while requests are ordered based on their arrival time: $r_{i,1}, \ldots, r_{i,j}, \ldots$.

5.1.1 Task WCET Estimation

Since the task can have different request types to the resource (e.g., reads vs. writes and misses vs. hits), a tighter bound on its cumulative DRAM access latency can be obtained by employing different latency bounds for each type of request [116, 44]. 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 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 $r_{i,j}$:

- RMP: Read miss requests to a private bank.
- 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 [12].
- 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.
- 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.

5.1.2 DuoMC Model

Figure 5.1 shows the conceptual architecture of DuoMC. Compared to Duetto we rename the RTA and HPA as the Real-Time command Scheduler (RTSch) and the High-Performance command Scheduler (HPSch), respectively, as they are used to schedule DRAM commands (discussed in Chapter 2). Following the Duetto reference model, DuoMC requires minimal modifications and knowledge about this HPSch. The latency guarantees provided by DuoMC, as well as the analysis detailed in Section 5.4, are completely independent of the internal behavior of the HPSch. In our implementation, the HPSch employs a FR-FCFS policy, as discussed in Chapter 2. It is crucial to point out that having two command schedulers does not pose additional challenges from an electrical/implementation point of

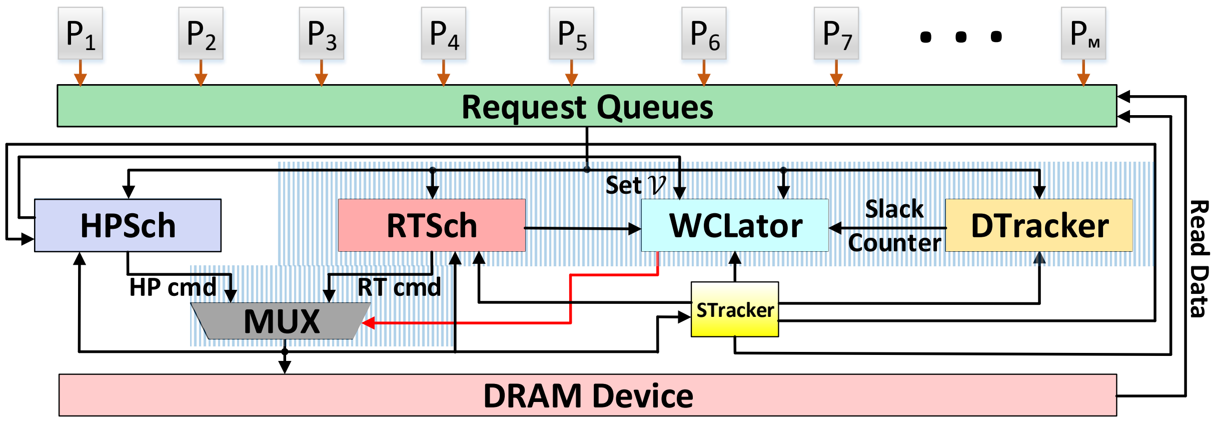


Figure 5.1: Conceptual architecture of DuoMC including four main components.

view, since both schedulers 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 5.1.

To address added complexities in managing DRAM, DuoMC further modifies the Duetto execution model. Specifically, in DRAM, the controller can issue a *CAS* command to a bank before it finishes processing the previous request; something that was not possible in the previous examples of Chapter 3 and 4. Because of this reason, we modify the execution model such that the finish time and completion time are differentiated. 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 Chapter 3, represents the time at which data is returned to the requestor (for a read). Note that this is different compared to the simple resource discussed in Chapter 3 model, where completion and finish time coincide. 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. The queueing and processing latency of a request are still defined based on arrival and finish times as in Chapter 3, because the stall time for a core depends on when the data is returned. However, the definition of outstanding request, oldest request, and the way the static WCL bound is computed must be updated to be based on the completion,

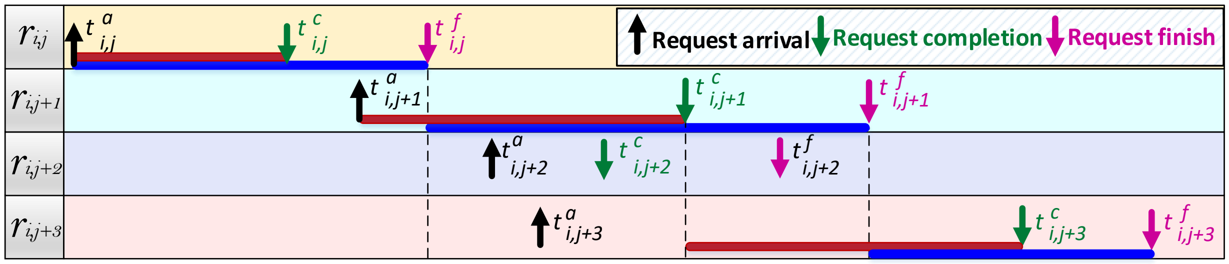


Figure 5.2: Processing latency shown in blue. Red bar represents the request being oldest. Assume that all previous requests $r_{i,k}$ with $k < j$ finish before $t^a_{i,j}$.

rather than finish time, because the controller has no control over the timing of the request once it completes. For the same reason, we assume that a request is put into the request queue once it arrives ($t^a_{i,j}$) and removed from the queue once it completes ($t^c_{i,j}$).

Definition 7 (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 8 (Outstanding and Oldest Request)**.** *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).*

Definition 9 (Static WCL Bound [75])**.** *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 DRAM 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.*

A clarifying example, similar to the one in Figure 3.3, is provided in Figure 5.2, 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. 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. Note that in the example in Figure 3.3, 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. Finally, the same Duetto guarantee on request deadlines as in Theorem 4 applies; specifically, the same reasoning as in the proof of Theorem 4 holds, except that we consider interval $[\max(t^c_{i,prec_j}, t^a_{i,j}), t^f_{i,j})$ when the request is oldest rather than $[\max(t^f_{i,prec_j}, t^a_{i,j}), d_{i,j})$.

A second change is relative to the behavior of the WCLator. In Chapter 3, we assumed that the WCLator considers all legal commands that can be issued by the HPA in the current cycle. In Figure 5.1, the WCLator receives as input every clock cycle a set of possible commands $\mathcal{V}$ provided by the HPSch. The key idea is that 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 5.5 has an extended discussion about these adaptations.

A final note is related to the refresh operations. As discussed in Chapter 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 the 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 has happened in the lifespan of the request) suffers an additional latency equal to the refresh overhead plus the deadline.

5.2 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. In order to improve the average-case performance, RTSch employs an open-page policy for the command generation. 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 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 [22]; 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 5.4 for a non-pipelined version of the controller. As we will discuss in Section 5.5, 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.

5.2.1 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.

5.2.2 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).

5.2.3 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.

5.2.4 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.

5.2.5 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.

5.2.6 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.

5.2.7 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 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.

5.2.8 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 [37], and the write requests to DRAMs in modern architectures are due to last-level cache evictions, and hence, they do not stall the pipeline [44, 121]. 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).

5.3 Illustrative Example for RTSch Rules

To illustrate the behavior of the RTSch, we next present two examples: Figure 5.3 focuses on the PRE and ACT arbiters according to Rules 1-4, while Figure 5.4 focuses on the CAS arbiter according to Rules 1, 5-7. The example in Figure 5.3 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

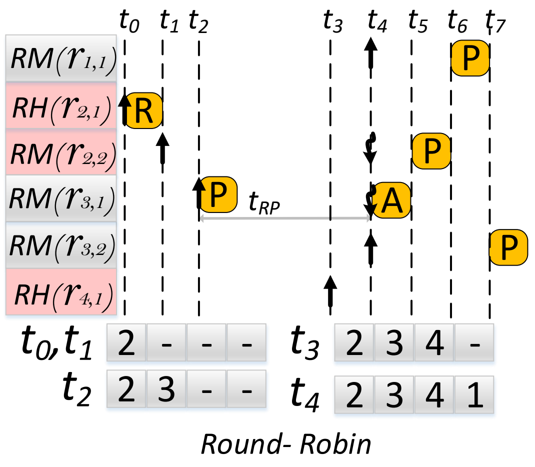


Figure 5.3: Illustrative example describing Rules 1-4 for ACT/PRE arbitration. Curly down arrows represent the time commands of a miss request become intra-ready.

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 5.4 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

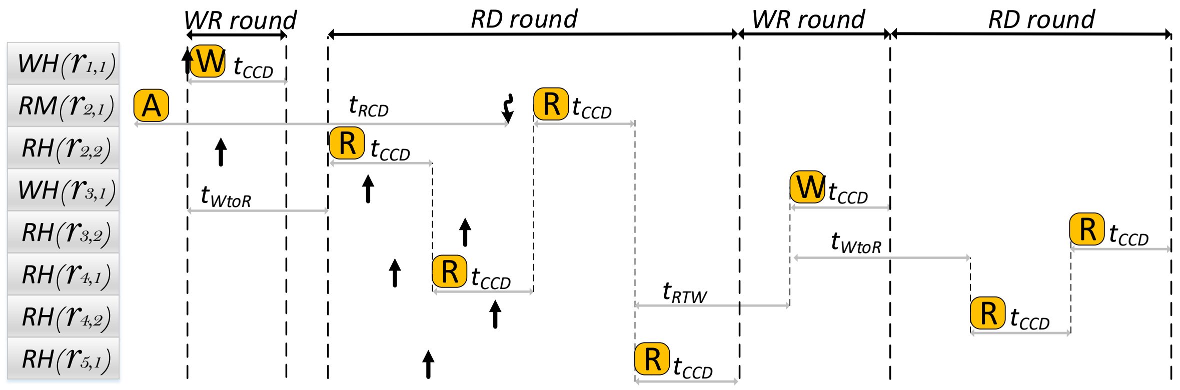


Figure 5.4: Illustrative example describing Rules 1, 5-7. Curly down arrows represent the time commands of a miss request become intra-ready.

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 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

Table 5.1: Symbols used in latency analysis.

Symbols	Description
P_i	Core i
b_k	Bank k
hp_i	Set of higher priority requestors than P_i
$\mathcal{S}$	The set of requestors in hp_i whose oldest request also targets b_r plus P_i
k^{PRE}	Number of PRE commands that can be issued ahead of $r_{i,j}$
k^{ACT}	number of ACT commands that can be issued ahead of $r_{i,j}$
k^{CAS}	Number of interfering requests
L^{PRE}	Maximum latency of PRE from the time it becomes intra-ready until it is issued
L^{ACT}	Maximum latency of ACT from the time it becomes intra-ready until it is issued
L^{RD}	Maximum latency of RD from the time it becomes intra-ready until it is issued
L^{RD}_{RD}	The latency bound of RD becomes intra-ready in read round
L^{RD}_{WR}	The latency bound of RD becomes intra-ready in write round
c^{inter}_{RD}	The number of cycles until a RD can be issued
c^{intra}_{RD}	The number of cycles until a RD becomes intra-ready
c^{intra}_{WR}	The number of cycles until a WR becomes intra-ready
c^{intra}_{PRE}	The number of cycles until a PRE becomes intra-ready
c^{intra}_{ACT}	The number of cycles until a ACT becomes intra-ready
$\Delta_i(RMP)$	The static latency bounds for a read miss request targeting a private bank
$\Delta_i(RHP)$	The static latency bounds for a read hit request targeting a private bank
$\Delta_i(MSq)$	The static latency bound for a miss request targeting a shared bank
$t^{Private}_{Residual}$	The worst-case latency from processing latency to PRE becoming intra-ready

at the back of the RR queue; hence, in the final read round, $r_{4,2}$ is serviced before $r_{3,2}$.

5.4 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 5.4.1, and to shared banks in Section 5.4.2; we focus on discussing key novel results. Then, we show how the static analysis can be modified to obtain the on-line WCLator estimation for the remaining processing latency in Section 5.4.3. Table 5.1 summarizes the symbols used in the latency analysis.

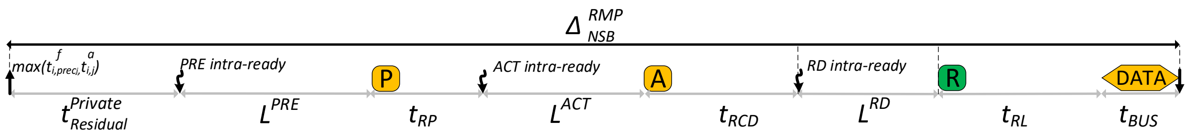


Figure 5.5: Request latency decomposition for read requests.

5.4.1 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 9, 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:

$$\Delta_i(RMP) \;=\; \max(\Delta^{RMP}_{SB}, \Delta^{RMP}_{NSB}). \tag{5.1}$$

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 5.5:

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;

3. L^{ACT} is the maximum latency of ACT from the time it becomes intra-ready until it is issued;

4. t_{RCD} is the ACT-to-CAS timing constraint;

5. L^{RD} is the maximum latency of RD from the time it becomes intra-ready until it is issued;

6. 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 5.6. 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 5.2.

$$t^{Private}_{Residual} = \max(t_{WR}, t_{RTP} - t_{RL} - t_{BUS}, t_{RAS} - \min(t_{RL}, t_{WL}) - t_{BUS} - 1). \tag{5.2}$$

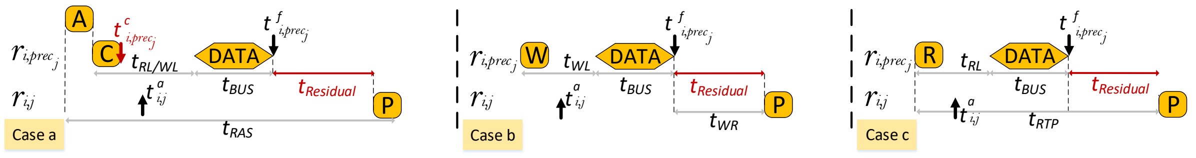


Figure 5.6: Case a, b, and c for the residual calculations for private bank access.

To facilitate the derivation of the on-line bounds in Section 5.4.3, 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 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 10. *While $r_{i,j}$ is oldest, the values of k^{PRE} and k^{ACT} cannot increase as long as the RTSch is selected.*

Observation 10 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 10, 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 5.7. Terms $\lceil \frac{L^{PRE}+1}{t_{RRD}} \rceil$ and $\lceil \frac{L^{PRE}+1}{t_{CCD}} \rceil$ in Equation 5.3 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 5.3.

$$L^{PRE}(k^{PRE}) = k^{PRE} + \left\lceil \frac{L^{PRE}(k^{PRE})+1}{t_{RRD}} \right\rceil + \left\lceil \frac{L^{PRE}(k^{PRE})+1}{t_{CCD}} \right\rceil. \qquad (5.3)$$

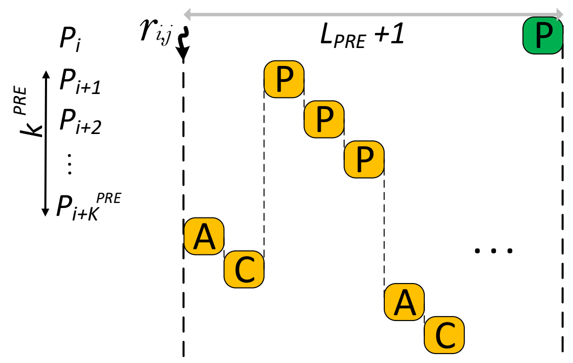


Figure 5.7: L_{PRE} example.

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 5.8. 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 5.4.

$$L^{ACT}(k^{ACT}) = t_{FAW} - 3 \cdot t_{RRD} + k^{ACT} \cdot (t_{RRD} + 1) + \left\lfloor \frac{k^{ACT}}{4} \right\rfloor \cdot (t_{FAW} + 1 - 4 \cdot t_{RRD} - 4) \tag{5.4}$$

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 $L^{ACT}(M - 1)$.

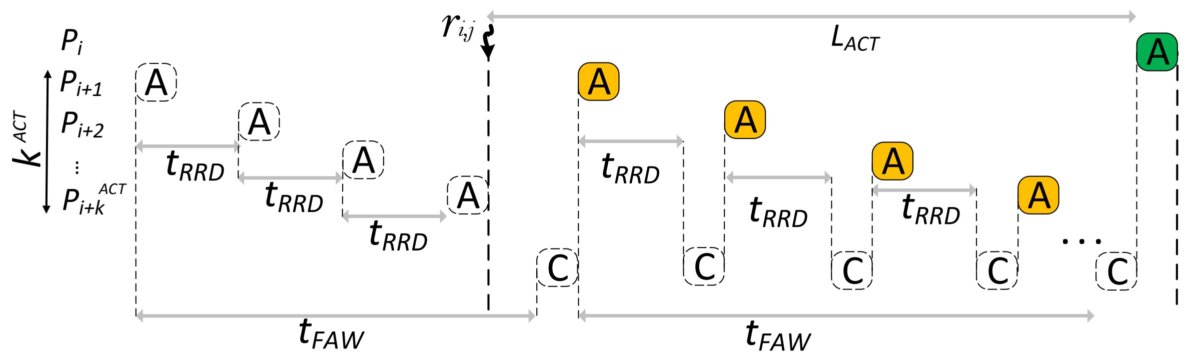


Figure 5.8: L_{ACT} example considering t_{RRD} and t_{FAW}.

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:

$$L^{RD}_{RD}(k^{RD}, c^{inter}_{RD}) = c^{inter}_{RD} + k^{RD} \cdot t_{CCD}. \tag{5.5}$$

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 $c^{inter}_{RD} = \max(t_{WtoR} - t_{CCD}, t_{CCD} - 1)$.

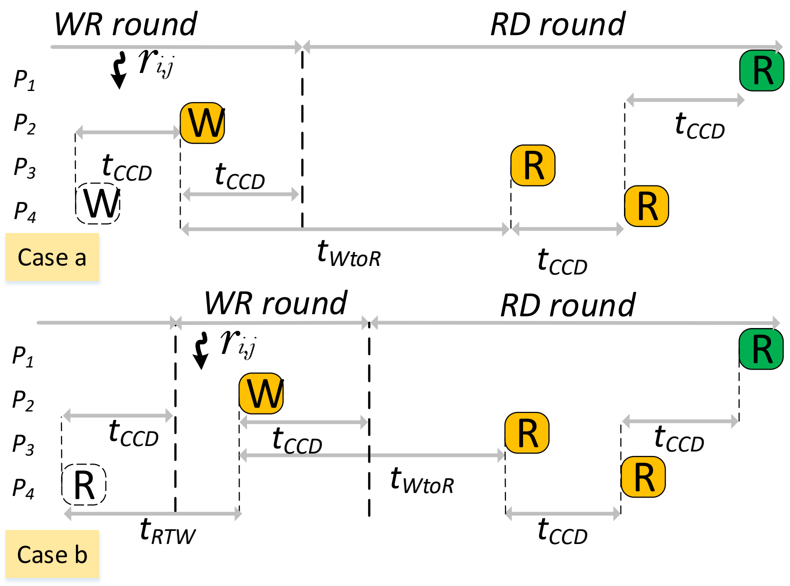


Figure 5.9: L^{RD}_{WR} example for $M = 4$ requestors, where $P_i = P_1$ and $k^{CAS} = 3$; in this example, two of the three interfering CAS commands are RD and one is WR.

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 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 5.9: (a) the WR command of a non-oldest request is issued immediately before the RD of $r_{i,j}$ becomes intra-ready, resulting in the following bound:

$$k^{CAS} \cdot t_{CCD} + t_{WtoR} - 1. \tag{5.6}$$

(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:

$$t_{RTW} - t_{CCD} + (k^{CAS} - 1) \cdot t_{CCD} + t_{WtoR} - 1 = \\ (k^{CAS} - 2) \cdot t_{CCD} + t_{RTW} + t_{WtoR} - 1. \tag{5.7}$$

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

$$L^{RD}_{WR}(k^{CAS}) = (k^{CAS} - 2) \cdot t_{CCD} + \max(t_{RTW}, 2 \cdot t_{CCD}) + t_{WtoR} - 1. \tag{5.8}$$

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 5.5 yields:

$$\Delta^{RMP}_{NSB} = t^{Private}_{Residual} + L^{PRE}(M-1) + t_{RP} + L^{ACT}(M-1) + \\ t_{RCD} + L^{RD}_{WR}(M-1) + t_{RL} + t_{BUS}. \tag{5.9}$$

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 the 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 5.10: 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:

$$
\begin{aligned}
\Delta_{SB}^{RMP} &= (M-1)\cdot t_{CCD} + t_{RTW} + (M-2)\cdot t_{CCD} + t_{WtoR} + \\
&\quad t_{RL} + t_{BUS} - (t_{RL} + t_{BUS}) \\
&= (2M-3)\cdot t_{CCD} + t_{RTW} + t_{WtoR}. \qquad (5.10)
\end{aligned}
$$

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:

$$\Delta_i(RHP) = \max(\Delta_{SB}^{RHP}, \Delta_{NSB}^{RHP}). \qquad (5.11)$$

Since the worst-case scenario for the self-blocking case in Figure 5.10 does not include the time for PRE and ACT commands, it also applies to a hit request, meaning that $\Delta_{SB}^{RHP} = \Delta_{SB}^{RMP}$. For the non-self-blocking case, we again decompose Δ_{NSB}^{RHP} into latency terms, similarly to Equation 5.9. 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:

$$\Delta_{NSB}^{RHP} = L_{WR}^{RD}(M-1) + t_{RL} + t_{BUS}. \qquad (5.12)$$

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

Write Latency. The derivation for the static latency bound $\Delta_i(WMP)$ of a write miss to private bank is similar to the one for the latency $\Delta_i(RMP)$ of read miss, except that read-related timing constraints are swapped for write-related timing constraints. Considering both the self-blocking and non-self-blocking cases, we have:

$$\Delta_i(WMP) \;=\; \max(\Delta_{SB}^{WMP}, \Delta_{NSB}^{WMP}). \qquad (5.13)$$

The latency decomposition for Δ_{NSB}^{WMP} is equivalent to the one in Figure 5.5, except that we consider a CAS latency term L^{WR} instead of L^{RD}, and a data sending time of $t_{WL} + t_{BUS}$.

Given that the worst-case CAS latency for WR can be found when the WR becomes intra-ready in a read round (L_{RD}^{WR}), this yields:

$$\Delta_{NSB}^{WMP} = t_{Residual}^{Private} + L^{PRE}(k^{PRE}) + t_{RP} + L^{ACT}(M-1) + t_{RCD} + L_{RD}^{WR}(M-1) + t_{WL} + t_{BUS}. \tag{5.14}$$

Since the case for L_{RD}^{WR} is symmetric to L_{WR}^{RD}, the same computation can be applied after switching read for write timing constraints in Equation 5.8:

$$L_{RD}^{WR}(k^{CAS}) = (k^{CAS} - 2) \cdot t_{CCD} + \max(t_{WtoR}, 2 \cdot t_{CCD}) + t_{RTW} - 1. \tag{5.15}$$

Similarly, computing the latency over Figure 5.10 after switching reads for writes and vice-versa yields the following bound for the self-blocking case Δ_{SB}^{WMP}:

$$\begin{aligned} \Delta_{SB}^{WMP} &= (M-1) \cdot t_{CCD} + t_{WtoR} + (M-2) \cdot t_{CCD} + t_{RTW} + t_{WL} + t_{BUS} - (t_{WL} + t_{BUS}) \\ &= (2M-3) \cdot t_{CCD} + t_{WtoR} + t_{RTW}, \end{aligned} \tag{5.16}$$

which is the same as the bound for Δ_{SB}^{RMP}. As in the case of read requests, this means that the non-self-blocking case has higher latency.

5.4.2 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 5.4.1, 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 $k_{\notin\mathcal{S}}^{PRE}, k_{\notin\mathcal{S}}^{ACT}, k_{\notin\mathcal{S}}^{RD}$ 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: $k_{\notin\mathcal{S}}^{PRE} \leq M - |\mathcal{S}|$ (the same holds for $k_{\notin\mathcal{S}}^{ACT}, k_{\notin\mathcal{S}}^{RD}$).

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_{First}^{MSq}; (2) the latency of the remaining $|\mathcal{S}|-1$ requests (including $r_{i,j}$ itself), which we denote as Δ_{Others}^{MSq}. 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

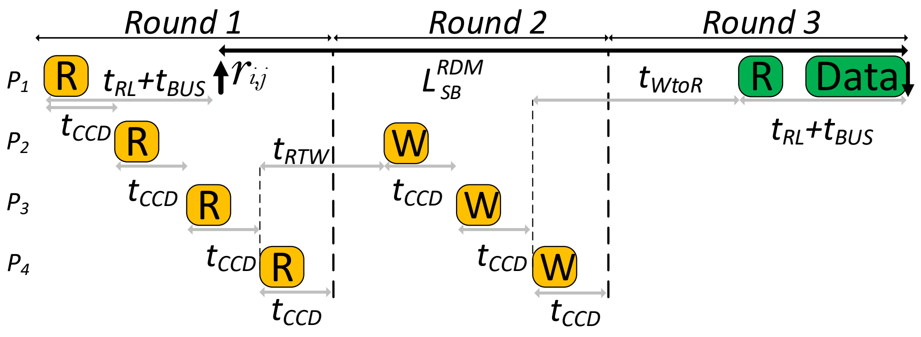


Figure 5.10: Worst-case scenario for a self-blocking read miss with $M = 4$.

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

$$L^{ACT}(k_1) + L^{ACT}(k_2) \leq L^{ACT}(k_1 + k_2) + L^{ACT}(0) \tag{5.17}$$

$$L^{PRE}(k_1) + L^{PRE}(k_2) \leq L^{PRE}(k_1 + k_2) + L^{PRE}(0) + 2 \tag{5.18}$$

Based on Observation 11, we can bound the PRE and ACT interference by simply assuming that the $k^{PRE}_{\notin \mathcal{S}}, k^{ACT}_{\notin \mathcal{S}}$ 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 5.8. 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:

$$\begin{aligned}\Delta^{MSq}_{First} \;=\;& t^{First}_{Residual} + L^{PRE}(k^{PRE}_{\notin\mathcal{S}}) + t_{RP} + L^{ACT}(k^{ACT}_{\notin\mathcal{S}}) + \\ & t_{RCD} + \max\big(L^{RD}_{WR}(M-|\mathcal{S}|) + t_{RL}, \\ & L^{WR}_{RD}(M-|\mathcal{S}|) + t_{WL}\big) + t_{BUS}, \end{aligned} \tag{5.19}$$

$$\begin{aligned}\Delta^{MSq}_{Others} \;=\;& \sum_{l=1}^{|\mathcal{S}|-1} \Big(t^{Others}_{Residual} + L^{PRE}(0) + 2 + t_{RP} + L^{ACT}(0) + \\ & t_{RCD} + \max\big(L^{RD}_{WR}(M-|\mathcal{S}|+l) + t_{RL}, \\ & L^{WR}_{RD}(M-|\mathcal{S}|+l) + t_{WL}\big) + t_{BUS}\Big). \end{aligned} \tag{5.20}$$

Note that L^{WR}_{RD} 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^{RD}_{WR} 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 to a private bank (Equations 5.9), 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 $t^{First}_{Residual}$ for the highest priority request, and $t^{Others}_{Residual}$ for the remaining $|\mathcal{S}|-1$ ones. $t^{First}_{Residual}$ is the worst-case latency from the start of processing latency $\max(t^{f}_{i,prec_j}, t^{a}_{i,j})$ of $r_{i,j}$ to the PRE of the highest priority requestor in $\mathcal{S}$ becoming intra-ready and $t^{Others}_{Residual}$ 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 $t^{Private}_{Residual}$ 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 $\max(t^{c}_{i,prec_j}, t^{a}_{i,j})$; while we know that only commands of requests in $\mathcal{S}$ can be issued afterwards. Hence, for the case of $t^{First}_{Residual}$, the worst-case scenario is that $t^{a}_{i,j} \geq t^{f}_{i,prec_j} > t^{c}_{i,prec_j}$ and either a CAS or ACT command is issued at cycle $t^{a}_{i,j}-1$, resulting in Equation 5.21. On the other hand, in the case of $t^{Others}_{Residual}$, 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 5.6, 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 5.11 shows the worst-case

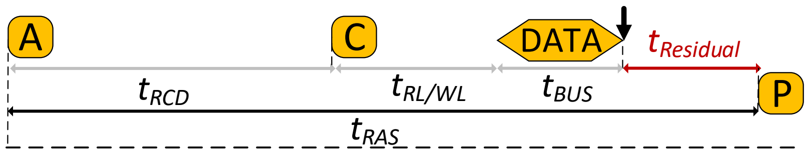


Figure 5.11: $t^{Others}_{Residual}$: computation of residual term for t_{RAS} constraint.

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 5.22.

$$t^{First}_{Residual} = \max(t_{WL} + t_{BUS} + t_{WR} - 1, t_{RTP} - 1, t_{RAS} - 1); \tag{5.21}$$

$$t^{Others}_{Residual} = \max(t_{WR}, t_{RTP} - t_{RL} - t_{BUS}, t_{RAS} - t_{RCD} - \min(t_{RL}, t_{WL}) - t_{BUS}). \tag{5.22}$$

Finally, we have to determine the value of $|\mathcal{S}|$ under which $\Delta_i(MSq) = \Delta^{MSq}_{First} + \Delta^{MSq}_{others}$ is maximized. This is non-trivial, since increasing $|\mathcal{S}|$ adds more terms in Equation 5.20, but it decreases the maximum value of $k^{PRE}_{\notin\mathcal{S}}, k^{ACT}_{\notin\mathcal{S}}$ 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 $|\mathcal{S}| = q$.

5.4.3 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 afterward. 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_{RD}^{inter} at time t, as well as the values of k^{PRE}, k^{ACT} and k^{RD} at time t, as defined in Section 5.4.1. 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 the 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 5.4.1. 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 $c_{PRE}^{intra} > 0$, 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_{PRE}^{intra} cycles, and following the latency decomposition for a non-self-blocking request in Figure 5.5, we can then bound the remaining latency as:

$$L_{online}^{RD,1} = c_{PRE}^{intra} + L^{PRE}(k^{PRE}) + t_{RP} + \\ L^{ACT}(k^{ACT}) + t_{RCD} + L_{WR}^{RD}(M-1) + t_{RL} + t_{BUS}. \tag{5.23}$$

  Equation 5.23 represents the base latency computation for Case (1); all other sub-cases use either the same or a modified computation of $L_{online}^{RD,1}$. **(1.1b)** If $c_{PRE}^{intra} = 0$, issuing a NOP can waste a clock cycle and increase the PRE delay by 1; hence the latency is $L_{online}^{RD,1} + 1$.

- **(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 5.23 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 5.3 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 5.23, 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 5.23 must be substituted with $\max(c^{intra}_{PRE}, t_{WL} + t_{BUS} + t_{WR})$.

Case (2): $b_r.row = -1$, then the RTSch needs to issue an ACT to row rw for $r_{i,j}$. The HPSch can only issue an ACT for bank b_r (to either row rw or another row), plus commands for other banks and NOP.

- **(2.1)** NOP: recall that in the worst case scenario used to compute $L^{ACT}(k^{ACT})$, the initial ACT is delayed by ACT commands sent before time t; hence, sending a NOP at time t does not change the bound, nor it changes the state of any bank, and the latency can be computed as:

$$L^{RD,2}_{online} = c^{intra}_{ACT} + L^{ACT}(k^{ACT}) + t_{RCD} + L^{RD}_{WR}(M-1) + t_{RL} + t_{BUS}. \tag{5.24}$$

- **(2.2)** PRE targeting bank $b_l \neq b_r$: this case is the same as NOP, so we use Equation 5.24. In addition, **(2.2a)** just like Case (1.2c), if at time t there exists requestor $P_q \in hp_i$ whose oldest request targets b_l and requires a CAS, then we increase k^{ACT} by one as the RTSch will need to now issue a PRE, ACT and CAS for that request.

- **(2.3)** ACT targeting bank $b_l \neq b_r$: **(2.3a)** if $c^{intra}_{ACT} > 0$, then issuing an ACT does not change the worst-case initial pattern leading to a delay of $t_{FAW} - 3 \cdot t_{RRD} - 1$; hence, we compute the latency as $L^{RD,2}_{online}$. **(2.3b)** If instead $c^{intra}_{ACT} = 0$, we use $L^{RD,2}_{online} + 1$ as the last ACT starting the t_{FAW} constraint is now issued in the current clock cycle, rather the the clock cycle before as in Figure 5.8. **(2.3c)** In addition to the two previous cases, if at time t there is a requestor $P_q \in hp_i$ whose oldest request targets b_l and the issued ACT opens the row targeted by such request, then we consider a value of k^{ACT} decreased by one when evaluating Equation 5.24.

- **(2.4)** CAS targeting bank $b_l \neq b_r$: as in Case (1.4) there is no extra command bus interference, nor the bank state changes, so the latency is $L^{RD,2}_{online}$.

- **(2.5)** ACT targeting bank b_r and row rw: as in Case (1.5), this is the ACT command required by $r_{i,j}$; then following the latency decomposition for $r_{i,j}$, the remaining latency is $t_{RCD} + L^{RD}_{WR}(M-1) + t_{RL} + t_{BUS}$.

- **(2.6)** ACT targeting bank b_r but a different row than rw: then the RTSch must first close bank b_r with a PRE before issuing an ACT and RD. Following the latency decomposition, and since the ACT-to-PRE intra-bank constraint is t_{RAS}, we obtain a latency of $t_{RAS} + L^{PRE}(k^{PRE}) + t_{RP} + L^{ACT}(k^{ACT}) + t_{RCD} + L^{RD}_{WR}(M-1) + t_{RL} + t_{BUS}$.

Case (3): $b_r.row = rw$, then the RTSch needs to issue a RD to row rw for $r_{i,j}$. The HPSch can select PRE and CAS for bank b_r, including the RD for $r_{i,j}$, plus commands for other banks and NOP.

- **(3.1)** NOP: the NOP could waste a cycle and delay the issuing of a RD command if $c_{RD}^{inter} = 0$. Hence, we can compute the latency bound as in the following equation, except that if $c_{RD}^{inter} = 0$, we set $c_{RD}^{inter} = 1$ to represent the NOP delay:

$$L_{online}^{RD,3} = \begin{cases} L_{RD}^{RD}(k^{RD}, c_{RD}^{inter}) & \text{if } c_{RD}^{intra} \leq 1 \\ c_{RD}^{intra} + L_{WR}^{RD}(M-1) & \text{if } c_{RD}^{intra} > 1 \end{cases} + t_{RL} + t_{BUS}. \qquad (5.25)$$

  Recall that the RTSch resets the round to read direction at time $t+1$, and it is always selected afterwards. Therefore, if $c_{RD}^{intra} \leq 1$, then the RD of $r_{i,j}$ is intra-ready at the beginning of the read round, and we can thus use the better RD latency bound L_{RD}^{RD}. Note we do not have to add c_{RD}^{intra} in this case even if $c_{RD}^{intra} = 1$ because c_{RD}^{inter} is at least 1, meaning the RD of $r_{i,j}$ could not be sent in this cycle anyway. If instead $c_{RD}^{intra} > 1$, then one or more WR commands could be issued before the RD of $r_{i,j}$ becomes intra-ready. Therefore, we cannot exclude the case where it becomes intra-ready during a write round, and we have to consider the worst-case RD latency bound L_{WR}^{RD}.

- **(3.2)** PRE targeting bank $b_l \neq b_r$: the same computation as for NOP applies.

- **(3.3)** ACT targeting bank $b_l \neq b_r$: again, the same computation as for NOP applies.

- **(3.4)** CAS targeting any bank, which does not serve $r_{i,j}$: we again use Equation 5.25, but in this case we need to set $c_{RD}^{inter} = t_{CCD}$ if a RD is issued, or $c_{RD}^{inter} = t_{WtoR}$ if a WR is issued. In addition, **(3.4a)** if a RD is issued which satisfies the oldest request of a requestor $P_q \in hp_i$, we subtract one from k^{RD} as that requestor will be removed from the RR queue.

- **(3.5)** PRE targeting bank b_r: similar to Case (2.6), the RTSch must reopen b_r to row rw with an ACT and then issue the CAS. Following the latency decomposition, we obtain a latency of $t_{RP} + L^{ACT}(k^{ACT}) + t_{RCD} + L_{WR}^{RD}(M-1) + t_{RL} + t_{BUS}$.

- **(3.6)** CAS targeting bank b_r and serving $r_{i,j}$: this is the RD command required by $r_{i,j}$, hence the remaining latency is exactly $t_{RL} + t_{BUS}$.

Next, we cover Case (3) for a write request. The command cases are the same as for the read case, but as discussed in Section 5.4.3, the base latency must be computed as:

$$L_{online}^{WR,3} = \max(c_{WR}^{intra}, t_{CCD}) + L_{RD}^{WR}(M-1) + t_{WL} + t_{BUS}. \qquad (5.26)$$

The $\max(c^{intra}_{WR}, t_{CCD}) + L^{WR}_{RD}(M-1)$ term is required because in the worst static case, we assume that the round switches from write to read the clock cycle before the write of $r_{i,j}$ becomes intra-ready, meaning that the WR was issued $t_{CCD}+1$ clock cycles in the past; however, in this case a WR could be issued the clock cycle before, triggering a t_{CCD} timing constraint.

Finally, we detail the case of a request to shared bank. As discussed in Section 5.4.3, the WCLator applies Cases (1)-(2)-(3) to the highest priority request in $\mathcal{S}$. A few modifications to the rules are required to account for the behavior of shared banks, as discussed below.

- Instead of parameters k^{PRE}, k^{ACT} and k^{RD}, both the latency equations and the rules that modify the value of such parameters before computing the equations use parameters $k^{PRE}_{\notin\mathcal{S}}, k^{ACT}_{\notin\mathcal{S}}$ and $k^{RD}_{\notin\mathcal{S}}$; this is safe since lower priority requests in $\mathcal{S}$ cannot interfere with the highest priority one.
- When considering Cases (2) and (3), if $r_{i,j}$ is not the highest priority request in $\mathcal{S}$, an additional latency term $L^{PRE}(k^{PRE}_{\notin\mathcal{S}}) - L^{PRE}(0)$ must be added; this is because PRE commands counted in $k^{PRE}_{\notin\mathcal{S}}$ can still interfere with other requests in $\mathcal{S}$, even if the highest priority one does not need to issue a PRE. For the same reason, when considering Case (3), a term $L^{ACT}(k^{PRE}_{o}) - L^{ACT}(0)$ must be added.
- Following the CAS latency computation used in Equation 5.19, we consider worst-case latencies of $L^{RD}_{WR}(M-|\mathcal{S}|)$ and $L^{WR}_{RD}(M-|\mathcal{S}|)$ instead of $L^{RD}_{WR}(M-1)$ and $L^{WR}_{RD}(M-1)$.
- The rules that involve issuing a command to the shared bank b_r need careful consideration, since the command could belong to either the highest priority requestor, or another requestor in $\mathcal{S}$, possibly $r_{i,j}$; we list and analyze all of them below.
- **(1.5)**: no change since a PRE command does not target any specific row in b_r.
- **(1.6)**: the part of the rule involving c^{intra}_{PRE} remains; in addition, if the CAS belongs to $r_{i,j}$, the latency is simply $t_{RL}+t_{BUS}$; while if it belongs to another requestor in $\mathcal{S}$ (which cannot be the highest priority for Case (1) to apply), we subtract one from $|\mathcal{S}|$.
- **(2.5)**: no change since this is the ACT required by the highest priority requestor in $\mathcal{S}$ (recall that we are estimating the remaining latency of the highest priority requestor, hence row rw is the row targeted by such requestor and not $r_{i,j}$).

- **(2.6)**: no change, since the RTSch will then give priority to the highest priority requestor in $\mathcal{S}$ and be forced to reopen its row.
- **(3.4)**: in addition to what is included in the rule, we have to cover the case in which the CAS targets b_r. If the CAS belongs to $r_{i,j}$, the latency is simply $t_{RL} + t_{BUS}$; while if it belongs to another requestor in $\mathcal{S}$ (which cannot be the highest priority one, otherwise Case (3.6) would apply here instead of (3.4)), we subtract one from $|\mathcal{S}|$.
- **(3.5)**: no change, since again a PRE command is not row-specific.
- **(3.6)**: no change since this is the CAS required by the highest priority requestor in $\mathcal{S}$.

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 5.4.2, 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 5.4.2).

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 a look-up table indexed based on the value of the various analysis parameters. We provide a more detailed implementation discussion in Section 5.5.

5.5 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 5.1 and used in Section 5.4) 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 5.4 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 same conservative design as in Chapter 3, where WCLator makes 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 5.4.3 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 5.12 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 5.7: 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 $D_i(\mathcal{T}(r_{i,j})) = \Delta_i(\mathcal{T}(r_{i,j}))$

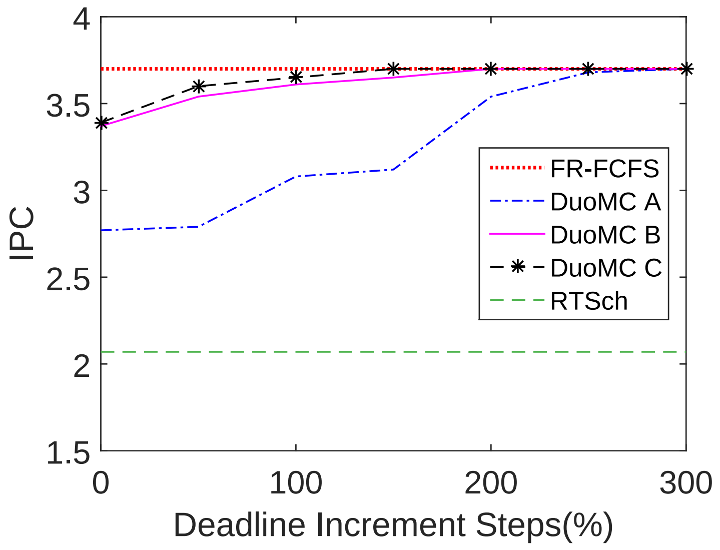


Figure 5.12: Overall IPC as a function of the relative deadline.

from the static analysis, and then progressively increase them up to $4 \times \Delta$ (300% increase). 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 5.4.3. 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 5.7 follows (B).

We next discuss how the three modules added by DuoMC to an existing COTS architecture (DTracker, RTSch and WCLator) 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 [22]. 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

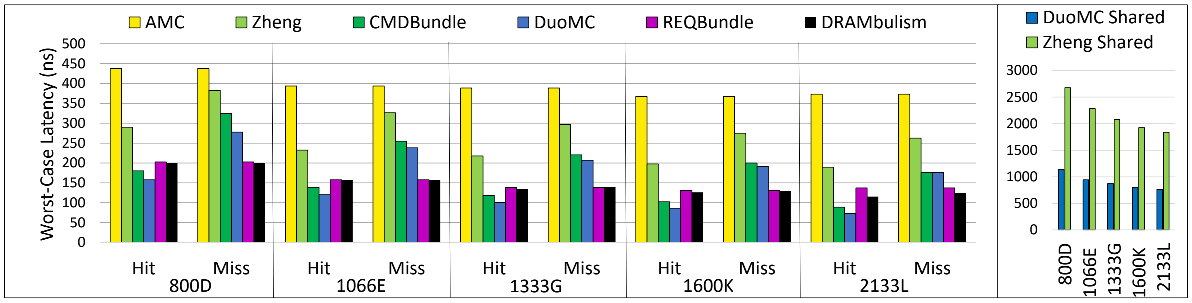


Figure 5.13: Analytical worst-case latency of read request (private and shared banks) across different speeds of DDR3 device.

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 [51, 92]. 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. If flushing the queues of RTSch is needed, the outdated commands of RTSch must be regenerated. Note that this operation is safe since as discussed earlier, we can take into account the time that RTSch requires to regenerate the commands and send them by increasing the bound by the number of stages - 1.

It remains to discuss the implementation of the WCLator. As shown in Section 5.4.3, 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 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 5.4.3, 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.

5.6 MCsim: An Extensible DRAM MC Simulator

In order to evaluate our proposed method DuoMC, we develop and release an open-source, extensible, memory controller simulator called MCsim [73]. In this section, we provide a detailed architectural overview of MCsim along with its configuration details.

5.6.1 Architectural Design

MCsim employs a modular, expansible, configurable, and integrable design; Figure 5.14 illustrates the major hardware blocks implemented in the framework. MCsim consists of an address translator (address mapping), which maps requests to physical memory cells, a command generator that converts requests into access commands, and request and command schedulers that determine the order of request/command execution.

Modularity: each block is constructed independently, and the encapsulated data is accessed through a simple interface. In this manner, changes to the behavior of a particular block do not impact the other blocks in the system. The specific algorithms implemented by these blocks must be customized based on the MC design. **Expansibility:** MCsim exploits the benefits of inheritance and polymorphism by providing virtual function interfaces, which minimize the amount of code required to extend the functionality of each block. **Configurability:** an MC simulator must include queues to connect the hardware blocks and temporarily store requests and commands. Rather than fixing the structure of the queues as in most other MC simulators, MCsim provides an easy to configure and modular queue structure. Since DRAM devices are organized in hierarchy levels (e.g.,

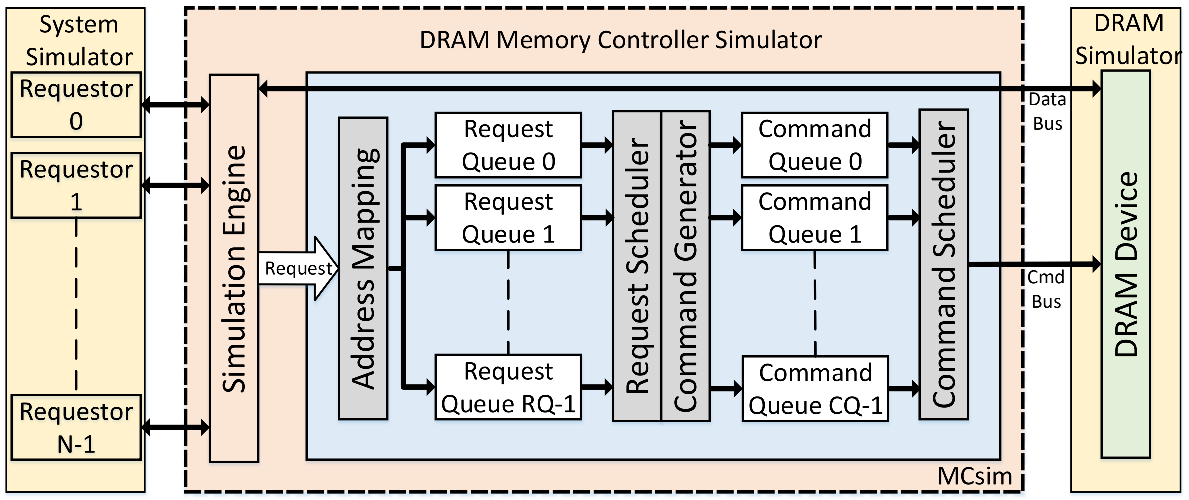


Figure 5.14: Generalized MC architecture and major blocks.

channels, ranks, bank groups, banks), the configurable queue structure allows the designer to construct them according to any DRAM level. **Integrability:** as shown in Figure 5.14, MCsim employs a generalized interface that can be accessed by any external system simulator to send memory requests and employs an abstract DRAM interface for the DRAM device model, so that the framework is not tied to any specific memory device type. For the device interface, we currently connect MCsim to Ramulator as the preferred device simulator since it supports a wide variety of DRAM standards. Note that a researcher can even implement his own device model and interface it directly to MCsim as far as it adheres to the MCsim interface. For the CPU system interface, MCsim can run as a trace-based simulator. It also provides an interface to connect two of the commonly used simulators, namely gem5 [11] and MacSim [61].

5.6.2 Configuration and Simulation Engine

A specific MC is built by a configuration file (`.ini`) to define the structure of the queues as well as the operation of each hardware block. As an example in Pseudo Code 5.1,

we show the configuration for the ORP controller [32], which requires per-requestor buffers and applies `DIRECT` request arbitration, `OPEN` command generation, and a specific `ORP` command scheduling policy.

MCsim also enables to configure the address mapping based on all possible different permutations of the DRAM device hierarchy, which allows the user to assign the mapping schemes flexibly. Each digit represents the corresponding hierarchy level, and the permutation determines the order of decoding. The permutation of the address bits can change the performance of a task based on how the data is allocated.

The request and command queues are constructed based on the selected DRAM hierarchy level. The bits value for each DRAM level is shown in lines 12 and 13 of Pseudo Code 5.1. These schemes are used to build the configured number of queues and also provide a flexible hardware structure to support most predictable DRAM scheduling policies.

```
1  // Rank[0],BankGroup[1],Bank[2],SubArray[3],Row[4],Col[5]
2  AddressMapping=012345 // order of address translation
3  RequestBuffer=0000    // request queue per level
4  ReqPerREQ=1           // request queue per requestor
5  WriteBuffer=0         // dedicated queue for write requests
6  CommandBuffer=0000    // command queue per level
7  CmdPerREQ=1           // command queue per requestor
8  // scheduler Based on Keys
9  RequestScheduler='DIRECT'// employ "DIRECT" request scheduler
10 CommandGenerator='OPEN'  // employ "OPEN" command generator
11 CommandScheduler='ORP'   // employ "ORP" command scheduler
12 // queue structure scheme: 0000 —> Channel, 1000 —> Rank
13 // 0100 —> BankGroup, 0010 —> Bank, 0001 —> SubArray
```

Pseudo Code 5.1: Configuration parameters for ORP

The structure of request and command queues is depicted in Figure 5.15. There are three separate buffers for request queues: first, a general buffer is used to store any incoming request; second, a set of buffers can be configured using the `ReqPerREQ` parameter to separate requests by individual requestors; and last, a write buffer can be enabled via the `WriteBuffer` parameter to separate write requests of any requestor from read requests. By providing these configurations, MCsim can support request schedulers that arbitrate among DRAM hierarchies, requestor IDs, type of requests, or all of the above. For example, some MCs schedulers arbitrate among requestors (cores) regardless of the DRAM location of a request [32]. Therefore, an individual buffer is created for each requestor. Other MCs arbitrate among DRAM banks rather than requestors and require a per-bank queue [52, 32]. The request arbitration can also be performed on two levels. For instance, DCmc requires a request queue per bank and performs RR among requestors in each bank. Typical high-performance arbiters employ First-Ready First-Come-First-Serve (FR-FCFS) arbitration in addition to a separate arbitration between read and write requests. The command queue shown in Figure 5.15 is similar to the request queue and can be configured based on two

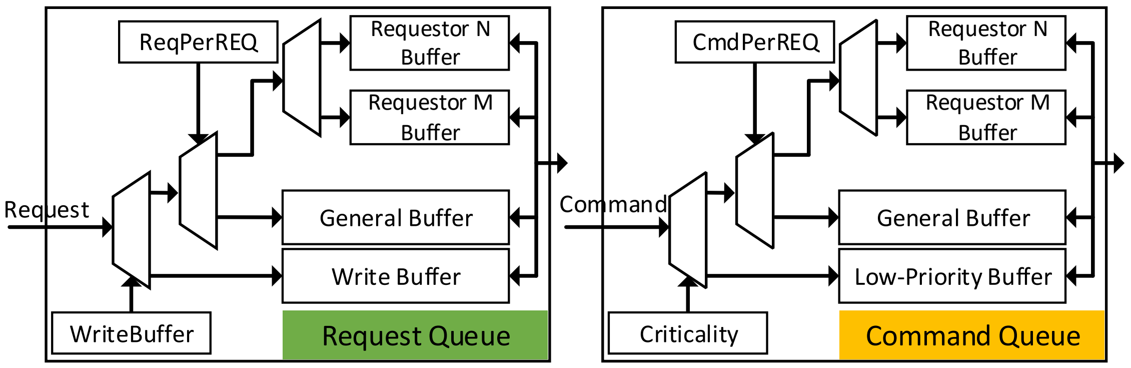


Figure 5.15: Request and command queue structures per-resource level.

parameters.

`CmdPerREQ` is used to separate commands for different requestors. Rather than having separate buffers for read and write requests, the command queue has separate command buffers for commands with different criticalities. Criticality reflects the priority of requests. Assigning different priorities for requests is a common theme both in predictable systems as well as COTS platforms. In a case of an MC with two priorities (for instance critical vs. non-critical requests), `General Buffer` is only employed for high requestors, while the `Low-Priority Buffer` stores lower-priority commands. The sub-classes of `RequestScheduler`, `CommandGenerator`, and `CommandScheduler` are selected based on their names. The string name of a subclass must be defined in `schedulerRegister.h` to notify which subclass will be used according to the names.

5.6.3 Detailed System Design

Throughout this section, we explain the detailed functionality and implementation of hardware blocks as well as their interactions according to the MCsim class diagram in Figure 5.16.

Top-Level Memory Controller: The top-level MemoryController is responsible for controlling the interaction between each internal hardware block and managing the requests and data flow between external memory requests and memory devices. In order to differentiate the requirements of each requestor in a system, we consider a requestor to

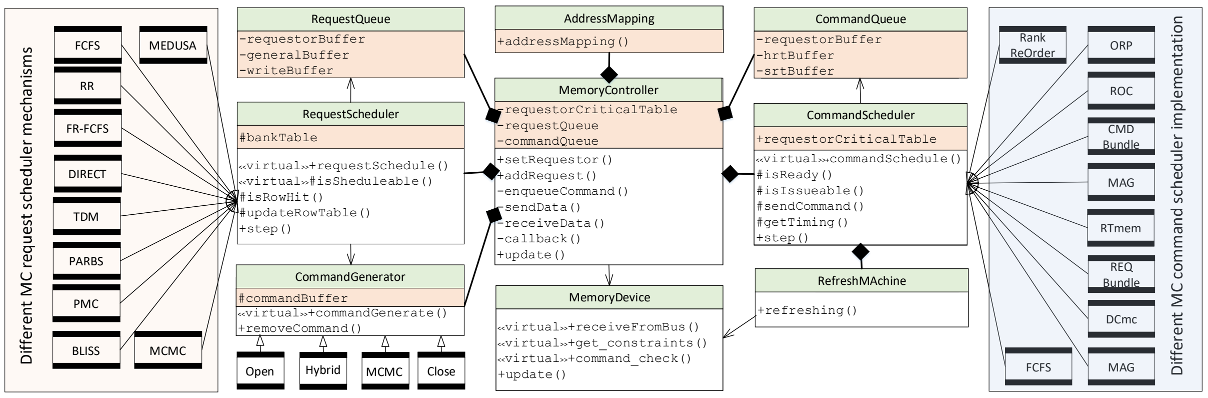


Figure 5.16: MCsim class diagram representing the main functional blocks in the simulator.

be either a critical (high priority) or non-critical (low priority).

Thus, a *requestorCriticalTable* can be configured by the user to indicate the criticality of each requestor. Then, the table can be used by any hardware block to make scheduling decisions among requests or commands based on the criticality of the requestors.

MemoryController receives new memory requests from requestors through `addRequest(ID, Address, Type, Size, Data)` and sends complete requests back to the requestors through `callback(Request)` provided by the simulation engine. MemoryController is also responsible for inserting requests and commands into their corresponding queues. Once there are available data that can be transmitted through the data bus, MemoryController communicates with the DRAM device through `receiveData(Data)` and `sendData(Data)` functions.

The `step()` function triggers the proceeding of each of the internal hardware blocks. According to Figure 5.16, the `requestSchedule()` function of RequestScheduler is first called to select requests that can be converted into commands. All commands generated by the CommandGenerator are en-queued into back-end command queues. Finally, the `commandSchedule()` function of CommandScheduler is called to issue an available command to the DRAM device. Since the data bus and the command bus are separated, available data can be sent or received in parallel with the command.

AddressMapping: The interface `addressMapping(request)` takes an incoming request and assigns a physical memory location to the request. The location of a request is determined by shifting the memory level bits in the order of the mapping scheme used in the configuration file.

RequestScheduler: The request scheduler is connected with both request and command queues because the arbitration may not only depend on the available requests in a request queue, but also on the status of corresponding command queues. For example, RTMem only allows a new request to be scheduled if there is no activate command in any of the command queues.

A *bankTable* is used to track the currently active row in the row buffer of each bank. It determines if a selected request is targeting an open or close row. The *bankTable* is accessed by `isRowHit(Request)` before a request is sent to command generator and updated by `updateRowTable(Rank,Bank,Row)` once the request is converted into commands.

CommandGenerator: The abstract class CommandGenerator has a virtual interface `commandGenerate(Request, isOpen)` which is called by requestScheduler to decode a request into a set of DRAM commands. This procedure is done based on the status of the row and generation pattern, including open, close, or hybrid page policies. All generated commands in one cycle are temporally stored in a command buffer, which is later accessed by the top-level memory controller. We separate requestScheduler and commandGenerator to allow development of each of those components separately.

CommandScheduler: The command scheduler is connected to the command queues and the DRAM device interface. It contains a `requestorCriticalTable` for each command queue to record the criticality of each requestor. `cmdQueueTimer` tracks the minimum number of clock cycles that any commands must wait before being issued. The table is updated once a command is issued to DRAM devices, and the counter for each command decremented every clock cycle. As an example, we show ORP scheduling mechanism in Pseudo Code 5.2. Ready commands from each requestor command buffer are first pushed to a First-In-First-Out (FIFO) buffer according to their criticality. When there is a CAS command in the critical FIFO that cannot be issued to the device, the CAS command will block all the other CAS commands (by *CASBlock*) in the FIFO, but not the commands of other types. If there is no command issuable from the high-priority FIFO, the scheduler tries to schedule a command from the low-priority FIFO.

MemorySystem: To support a broad range of DRAM standards, MCsim has a general interface to access DRAM information provided by the user through three virtual functions: 1) `GET_CONSTRAINT(name)`: the MC retrieves the timing constraint values for a selected DRAM device from the device simulator. 2) `CHECK_COMMAND(Cmd)`: once a command is selected from the command scheduler, this function is used to determine whether the command can be issued in the current clock; 3) `RECEIVE_COMMAND(Cmd)`: behaves as an interface to the command bus; it is called by `sendCommand(Cmd)` in commandScheduler to issue a command through the command bus. Notice that, if the implementation of a certain MC design requires changes in the device itself, some modifications also need to be done in MCsim which would be straightforward due to the modularity of the simulator.

```
Function scheduleCommand_ORP()
    for('each requestorBuffer in a channel commandQueue')
        if('requestorBuffer is not empty')
            get front Cmd from the requestorBuffer;
                if('isReady(Cmd)')
                    if('HRT requestor')
                        push Cmd into FIFO;
                    else
                        push Cmd into SRT-FIFO;
    CASblock = false;
    for('every Cmd in FIFO from the front of the queue')
        if('CASblock is true and Cmd is CAS')
            continue;
            if('issIssueable(Cmd)')
                sendCommand (Cmd);
                return Cmd;
            else if('Cmd is CAS')
                CASblock = true;
    for('every Cmd in the SRT-FIFO from front of the queue')
        if('isIssueable(Cmd)')
            sendCommand(Cmd);
            return Cmd;
    return NULL;
```

Pseudo Code 5.2: ORP command scheduler

5.6.4 MCsim Evaluation and Validation

1) **Lines-of-Code (LOC):** To show the effectiveness of MCsim we implement 10 different MC scheduling techniques at the request-level and 11 ones at the command-level as Figure 5.16 illustrates. We observe that the maximum amount of controller-specific code in MCsim is less than 11%, and the largest amount of code required to implement any controller is 432. In Table 5.2, we report the LOC required to implement each MC. In addition, we also implemented a device-based refresh mechanism (per-bank refresh [16]) that only adds 65 LOC to MCsim.

2) **Simulation Time (RunTime):** We compare MCsim with existing DRAM and MC simulators, which are open-sourced and can run as a standalone package with inputs trace,

Table 5.2: Simulation time (sec) of MCs for different simulators. ✓represents the ability to distinguish among different requestors in each MC.

Controller	REQ	Simulator	LOC	$RunTime_{seq}$	$RunTime_{rand}$
FR-FCFS	✗	MCsim	32	3.91	8.7
	✗	Ramulator#	309	18.97	31.01
	✗	Ramulator	243	6.04	9.12
	✗	DRAMsim2	356	3.28	8.28
BLISS	✓	MCsim	78	32.01	75.71
	✓	Ramulator#	335	293.32	422.52
PARBS	✓	MCsim	138	46.95	115.66
	✓	Ramulator#	424	172.62	372.21
ORP	✓	MCsim	93	44.80	74.83
	✓	Standalone	542	103.76	726.31
RTMem	✓	MCsim	96	86.69	98.16
	✓	Standalone	1910	50.12	51.24
MEDUSA	✓	MCsim	123	33.78	98.30
CMDBundle	✓	MCsim	169	37.95	73.27
REQBundle	✓	MCsim	432	52.27	57.49
MAG	✓	MCsim	94	40.31	71.16
MCMC	✓	MCsim	182	63.61	66.95
DCmc	✓	MCsim	85	42.81	71.35
AMC	✓	MCsim	98	81.90	92.38
PMC	✓	MCsim	65	93.97	105.52
ROC	✓	MCsim	175	40.38	72.54

making it viable to provide a fair and reproducible comparison. We employ two synthetic memory traces containing one million requests each, including 90% read and 10% write requests since reads are more critical in general. The `seq` trace is constructed such that it accesses rows consecutively, which tends to access open rows in the device. The `rand` trace is created with completely randomized address locations in order to stress the controllers with close requests. Since DDR3 device is supported in all the simulators, we run each simulator on our host (Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, 16GB RAM, and the Linux kernel is 4.15) with DDR3 1600H device with the same DRAM structures and timing constraints and make sure each simulator has the same system parameters. We configured all device simulators to employ FR-FCFS scheduling. The simulation time are shown in Table 5.2. Regarding FR-FCFS, the run time is presented, such that each simulator finishes all the requests in the trace. For the rest of the controllers that are concerned about fairness, we duplicate the trace into eight such that each requestor accesses its own trace, which consists of 1 million requests as used previously. We conclude that MCsim provides comparable simulation time to the device simulators, as well as the greatest extensibility that enables us to develop new controllers. Notice that the extra lines of code for implementing a new controller is small; however, on the downside, the generalization feature of MCsim might slow down the simulation speed (specifically, this is the case for RTMem).

3) **Behavioral Validation:** We validate MCsim using the regression test suite provided by the DRAM subsystem validation tool, MCXplore [41]. This regression suite covers a wide range of controller parameters such as hit ratio, read/write ratio, and interleaving. We validate the correctness of MCsim by comparing the issuing times of commands with the JEDEC standard's dictated constraints. We also compare each policy with its counterpart in a publicly available simulator as Table 5.2 shows. Results confirm that MCsim conforms to the standard and the behavior of each policy matches that of the corresponding simulator.

5.7 Evaluation

Experimental Setup. We use MacSim [60], 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 [74], 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 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 [97, 81], 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) [84], 2) CMDBundle [22], 3) DRAMbulism [74], 4) REQBundle [33], and 5) Zheng [117]. For ease of comparison, we do not model refresh operations.

Workloads. We use EEMBC 1.1 auto benchmark suite [91] and also use two synthetic benchmarks: latency and bandwidth from IsolBench [110]. 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.

5.7.1 Analytical Worst-Case Memory Access Latency

Figure 5.13 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 [117] 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 [84] 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 [22] 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 [74] and REQBundle [33] 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.

5.7.2 Measured Request Latency

Figure 5.17 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 5.17 only incorporates requests with latency longer than 80 cycles. The black dashed line represents the static WCL bound for DuoMC. For the FR-FCFS controller, we observe noticeable latency 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.

5.7.3 Average-Case Performance

Figure 5.18a 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 $2\times$ of the minimum deadline (100% increase). Notice that, the deadline setting for each requestor is fully configurable and can be determined based on the requirements/characteristics of the application/requestor (i.e. existing slacks,

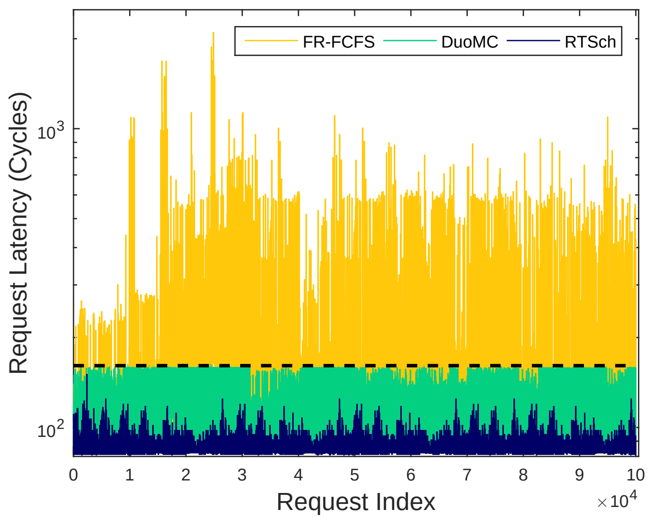


Figure 5.17: Request latency comparison amongst DuoMC, FR-FCFS, and RTSch. Only latencies greater than 80 cycles are shown. Note that Y-axis represents the latency in log_{10} scale.

criticality of the requestor, or criticality of different partitions of the task). 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 the 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 time. In Figure 5.19, DuoMC shows an even better relative performance with DuoMC-0 compared to the Figure 5.18, resulting in only 7% slowdown compared to HPSch. For comparison,

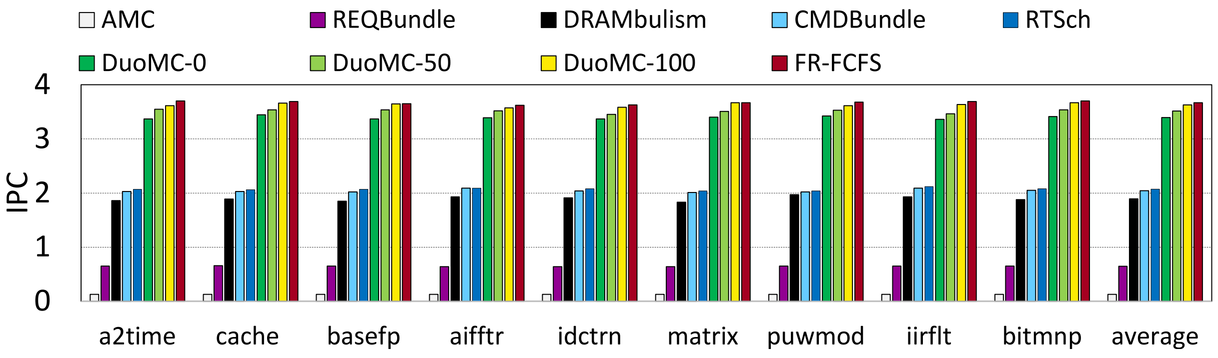


Figure 5.18: IPC for EEMBC benchmarks using DDR3-1600K.

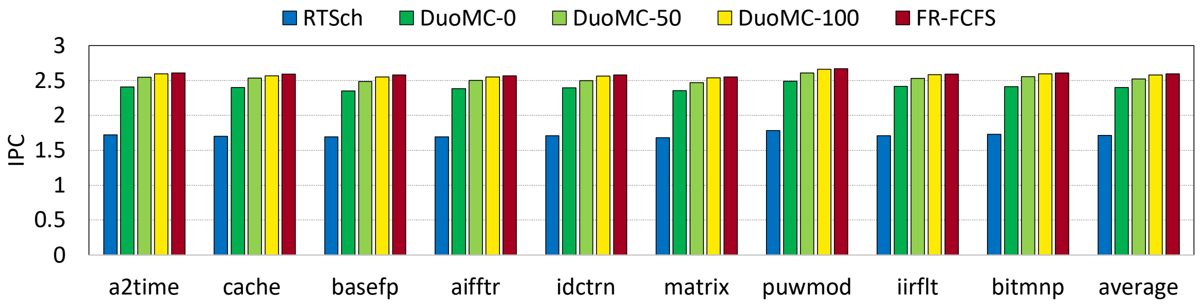


Figure 5.19: IPC for EEMBC benchmarks using DDR4-2400U.

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 [26]; 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.

5.8 Summary

In this chapter, we introduced DuoMC, a novel MC to manage DRAM memories in high-performance real-time embedded systems. DuoMC embodies two main contributions: 1) It applies Duetto reference model to COTS DRAM controllers as one of the most complex shared resources. 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.

Chapter 6

Conclusion and Future Work

Real-time embedded systems are experiencing a transformation towards incrementally more integrated architectures. Multi-core platforms, heterogeneous co-processors, DMA engines, multi-level caches, and superscalar execution are employed to appease accrescent performance demands through increasing the complexity of the system. These complexities expose a significant amount of challenges in terms of interference amongst requestors that compete to access shared resources. This challenge creates a fundamental trade-off between average-case performance and predictability of the system.

Based on the observation that the system is not overloaded most of the time, this thesis presents Duetto reference model that successfully addresses the trade-off mentioned above. This can be achieved by introducing a dual-mode arbitration scheme and run-time monitoring the request latencies such that predictability is achieved by design while the impact on the system performance is negligible.

This dissertation's research contributions elucidate generalized steps for the real-time system designers to address the performance-predictability trade-off for the shared resources in multi-core platforms. The key contributions of this thesis include:

1. A generalized reference model for hardware resource management is introduced that tackles the predictability-performance trade-off by pairing a real-time arbiter with a high-performance arbiter. Duetto tends to execute the high-performance arbiter most of the time but is able to switch to the real-time arbiter only in the rare cases when timing guarantees risk being violated.

2. A high-performance yet predictable hardware cache coherency arbiter is introduced that beats the state-of-the-art predictable coherency work in terms of average performance.

3. The proposed cache coherency arbitration mechanism is further improved by integration with Duetto reference model. It is shown that by leveraging Duetto and using run-time information of the system, it is possible to achieve a performance similar to COTS-based mechanisms.

4. A highly optimized DRAM memory controller design is proposed that provides tight latency bounds. Moreover, a near-COTS performance is achieved by employing Duetto. The proposed scheme supports shared banks which was not possible in previous works or was doable with significantly pessimistic bounds.

5. An open-source, extensible, memory controller simulator called MCsimis developed and released to simulate DRAM memory controller. MCsim is able to evaluate the performance of the memory controllers. Moreover, MCsim can run in both standalone mode as well as full system simulation mode through integration with MACsim [61].

6. This thesis clarifies that different resources bring different complexities. In the case of DRAM, the complexity is sending commands before the previous request is finished. In addition, the resource states are not kept in the resource itself and there needs to be some sort of estimator. In the case of cache, there are more cases in terms of the sequence of operation each request could follow in the system which depends on the state of a cache line and the coherency state machine.

The techniques proposed in this thesis enable multiple future extensions based on the existing limitations. This includes:

1. In reality, the resources described in Chapter 4 and Chapter 5 interact with each other. There exists a real-time arbiter for each of them but the latency analysis is performed separately. Since in practice there is a pipeline of requests over these two resources, doing a combined latency analysis that considers the interaction between the resources might lead to a tighter bound rather than summing the latency of each of these two resources. In detail, Chapter 4 assumed that the LLC is a perfect cache such that the delay imposed by accessing the off-chip memory is not modeled. However, in a more realistic model, a particular request could miss in the last on-chip memory hierarchy. This miss request then will be directed to the memory controller.

2. Incorporate Duetto in all memory hierarchy levels including memory controller and LLC together and considering their interaction. This requires a more complex/detailed latency analysis as well as synchronization. In detail, specific challenge is how to properly design Duetto to interact across multiple resources. For example, we expect that there might be trade-off in whether a unique arbiter is running over two resources which might lead to better bound but more complex implementation or whether different Duetto(s) should be implemented but keeping track of the deadlines and RR order might lead to a more pessimistic bound.

3. One of the complexities of Duetto is deriving the on-line WCL analysis. While in this thesis, we do so manually, we believe that a significant portion of the process could be (partially) automated through either model-checking or computer-assisted proofs; note that the set of analysis cases can be trivially enumerated by a tool based on the resource state machine.

4. The mechanisms introduced in this thesis assumed all cores in the system are homogeneous meaning that their requirements are similar. However, in a real world SoC, some IPs have different requirements (e.g. display engine which needs to meet the bandwidth to fill the display frame buffer and has latency requirements to not miss pixels in a frame). Or in a more specific way, GPUs consume a significant amount of data while processing cores might not act similarly. Duetto is configurable such that different requestors in the system could have different latency bounds (deadlines). It would be interesting to conduct further simulation using integrated heterogeneous simulators and investigate the impact of varying deadlines on the performance of processing components.

5. The evaluation in this thesis is performed through architectural simulators. While we tried our best to engineer the design such that it runs fast enough (specifically WCLator design), there still needs to be a concrete implementation to prove the design satisfies the performance requirements. I envision that the reference model proposed in this thesis works better as an "add-on" to an already existing hardware component compared to re-designing a new architecture.

6. As discussed in Chapter 5, most of these designs are pipelined but the architectural simulators do not support pipeline behavior. Hence, it would be interesting to explore what would happen by fully pipelining the design and implementing it on the actual RTL for example starting a proof of concept on FPGA.

7. In Duetto reference model, we incorporate a request-driven approach meaning that we compute the interference on the shared resource by considering the worst-case scenario that could happen to each and every request. However, incorporating more knowledge about the other requestors in the system could result in a better bound (i.e. a job-driven approach). For example, the system could track the latencies based on multiple requests through set of counters that count the number of interfering requests of other cores. However, this would add extra complexity and require additional information on the operation of the other requestors in the system.

References

[1] Benny Akesson and Kees Goossens. *Memory controllers for real-time embedded systems.* Springer, 2011.

[2] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable SDRAM memory controller. In *IEEE/ACM international conference on Hardware/-software codesign and system synthesis*, pages 251–256, 2007.

[3] ARM Cortex-A9 - technical reference manual.

[4] ARM. Amba axi protocol specification v2. 0. *ARM Holdings*, 2010.

[5] Ayoosh Bansal, Jayati Singh, Yifan Hao, Jen-Yang Wen, Renato Mancuso, and Marco Caccamo. Cache where you want! reconciling predictability and coherent caching. *arXiv preprint arXiv:1909.05349*, 2019.

[6] Ayoosh Bansal, Jayati Singh, Yifan Hao, Jen-Yang Wen, Renato Mancuso, and Marco Caccamo. Reconciling predictability and coherent caching. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–6. IEEE, 2020.

[7] Sanjoy Baruah and Steve Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *2008 Euromicro Conference on Real-Time Systems*, pages 147–155. IEEE, 2008.

[8] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24. IEEE, 2016.

[9] Guillem Bernat, Antoine Colin, and Stefan M Petters. Wcet analysis of probabilistic hard real-time systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 279–288. IEEE, 2002.

[10] Balasubramanya Bhat and Frank Mueller. Making dram refresh predictable. *Real-Time Systems*, 47(5):430–453, 2011.

[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

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

[13] Alan Burns and Robert Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, pages 1–69, 2013.

[14] Jason F Cantin, Mikko H Lipasti, and James E Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 246–257. IEEE, 2005.

[15] Jason F Cantin, James E Smith, Mikko H Lipasti, Andreas Moshovos, and Babak Falsafi. Coarse-grain coherence tracking: Regionscout and region coherence arrays. *IEEE Micro*, 26(1):70–79, 2006.

[16] Kevin Kai-Wei Chang, Donghyuk Lee, Zeshan Chishti, Alaa R Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. Improving dram performance by parallelizing refreshes with accesses. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 356–367. IEEE, 2014.

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

[18] Bekim Cilku, Bernhard Frömel, and Peter Puschner. A dual-layer bus arbiter for mixed-criticality systems with hypervisors. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 147–151. IEEE, 2014.

[19] Antoine Colin and Stefan M Petters. Experimental evaluation of code properties for wcet analysis. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 190–199. IEEE, 2003.

[20] Tanya L Crenshaw, Elsa Gunter, Craig L Robinson, Lui Sha, and PR Kumar. The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures. In *IEEE International Real-Time Systems Symposium (RTSS)*, 2007.

[21] Anup Das, Hasan Hassan, and Onur Mutlu. Vrl-dram: improving dram performance via variable refresh latency. In *DAC*, volume 3, page 2, 2018.

[22] Leonardo Ecco and Rolf Ernst. Improved DRAM Timing Bounds for Real-Time DRAM Controllers with Read/Write Bundling. In *Real-Time Systems Symposium (RTSS)*, pages 53–64, 2015.

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

[24] Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2014.

[25] Michael A Fischer. Fair arbitration technique for a split transaction bus in a multiprocessor computer system, November 15 1988. US Patent 4,785,394.

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

[27] Sven Goossens, Benny Akesson, and Kees Goossens. Conservative Open-page Policy for Mixed Time-Criticality Memory Controllers. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 525–530. EDA Consortium, 2013.

[28] Sven Goossens, Jasper Kuijsten, Benny Akesson, and Kees Goossens. A reconfigurable real-time sdram controller for mixed time-criticality systems. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, page 2. IEEE Press, 2013.

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

[30] Giovani Gracioli and Antônio Augusto Fröhlich. On the design and evaluation of a real-time operating system for cache-coherent multicore architectures. *ACM SIGOPS Operating Systems Review*, 49(2):2–16, 2016.

[31] Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. 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, 2019.

[32] Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A comparative study of predictable dram controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):1–23, 2018.

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

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

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

[36] Mohamed Hassan. On the off-chip memory latency of real-time systems: Is ddr dram really the best option? In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

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

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

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

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

[41] Mohamed Hassan and Hiren Patel. Mcxplore: Automating the validation process of dram memory controller designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(5):1050–1063, 2017.

[42] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. A framework for scheduling dram memory accesses for multi-core mixed-time critical systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 307–316. IEEE, 2015.

[43] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. PMC: A requirement-aware DRAM controller for multi-core mixed criticality systems. In *ACM Transactions on Embedded Computing Systems (TECS)*, 2016.

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

[45] Mohamed Hassan and Rodolfo Pellizzoni. Analysis of memory-contention in heterogeneous cots mpsocs. In *Euromicro Conference on Real-Time Systems*, 2020.

[46] Farouk Hebbache, Mathieu Jan, Florian Brandner, and Laurent Pautet. Shedding the shackles of time-division multiplexing. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

[47] Sven Heithecker and Rolf Ernst. Traffic shaping for an fpga based sdram controller with complex qos requirements. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 575–578. IEEE, 2005.

[48] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[49] Salah Hessien and Mohamed Hassan. The Best of All Worlds: Improving Predictability at the Performance of Conventional Coherence with No Protocol Modifications. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, October 2020.

[50] Intel® 64 and ia-32 architectures optimization reference manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf. Accessed: 2021-07-20.

[51] Intel. External memory interface handbook volume 2: Design guidelines, 2017.

[52] Bruce Jacob, David Wang, and Spencer Ng. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

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

[54] Manolis Katevenis, Stefanos Sidiropoulos, and Costas Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose atm switch chip. *IEEE Journal on selected Areas in Communications*, 9(8):1265–1279, 1991.

[55] Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. Designing predictable cache coherence protocols for multi-core real-time systems. *IEEE Transactions on Computers*, 2020.

[56] Anirudh Mohan Kaushik and Hiren Patel. A systematic approach to achieving tight worst-case latency and high-performance under predictable cache coherence. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 105–117. IEEE, 2021.

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

[58] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. In *2011 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–12. IEEE, 2011.

[59] Hokeun Kim, David Broman, Edward A Lee, Michael Zimmer, Aviral Shrivastava, and Junkwang Oh. A predictable and command-level priority-based dram controller for mixed-criticality systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 317–326. IEEE, 2015.

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

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

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

[63] Namhoon Kim, Micaiah Chisholm, Nathan Otterness, James H Anderson, and F Donelson Smith. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 223–234. IEEE, 2017.

[64] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[65] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 65–76. IEEE Computer Society, 2010.

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

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

[68] Benjamin Lesage, Isabelle Puaut, and André Seznec. Preti: Partitioned real-time shared cache for mixed-criticality real-time systems. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, pages 171–180, 2012.

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

[70] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[71] Renato Mancuso, Rodolfo Pellizzoni, Neriman Tokcan, and Marco Caccamo. Wcet derivation under single core equivalence with explicit memory budget assignment. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[72] Peter Marwedel. *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer Science & Business Media, 2010.

[73] Reza Mirosanlou, Danlu Guo, Mohamed Hassan, and Rodolfo Pellizzoni. Mcsim: An extensible dram memory controller simulator. *IEEE Computer Architecture Letters (CAL)*, 2020.

[74] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Drambulism: Balancing performance and predictability through dynamic pipelining. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[75] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Duetto: Latency guarantees at minimal performance cost. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1136–1141, 2021.

[76] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Duomc: Tight dram latency boundswith shared banks and near-cots performance. In *2021 ACM International Symposium on Memory Systems (MEMSYS)*, 2021.

[77] Anastasio Molano, Kanaka Juvva, and Ragunathan Rajkumar. Real-time filesystems. guaranteeing timing constraints for disk accesses in rt-mach. In *Proceedings Real-Time Systems Symposium*, pages 155–165. IEEE, 1997.

[78] Shubhendu S Mukherjee, Babak Falsafi, Mark D Hill, and David A Wood. Coherent network interfaces for fine-grain communication. *ACM SIGARCH Computer Architecture News*, 24(2):247–258, 1996.

[79] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *2008 International Symposium on Computer Architecture*, pages 63–74. IEEE, 2008.

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

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

[82] Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 109–118. IEEE, 2014.

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

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

[85] Marco Paolieri, Eduardo Quiñones, and Fransisco J. Cazorla. Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions. *Transactions on Embedded Computing Systems (TECS)*, 2013.

[86] Risat Mahmud Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 309–320. IEEE, 2012.

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

[88] Rodolfo Pellizzoni and Marco Caccamo. Impact of peripheral-processor interference on wcet analysis of real-time embedded systems. *IEEE Transactions on Computers*, 59(3):400–415, 2010.

[89] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 741–746. IEEE, 2010.

[90] Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, 1995.

[91] Jason Poovey. Characterization of the EEMBC benchmark suite. *North Carolina State University*, 2007.

[92] Qualcomm. Qualcomm snapdragon 600e processor apq8064e recommended memory controller and device settings application note, 2016.

[93] Moinuddin K Qureshi, Dae-Hyun Kim, Samira Khan, Prashant J Nair, and Onur Mutlu. Avatar: A variable-retention-time (vrt) aware refresh for dram systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 427–437. IEEE, 2015.

[94] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. PRET DRAM controller: bank privatization for predictability and temporal isolation. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis*, CODES+ISSS '11, pages 99–108, New York, NY, USA, 2011. ACM.

[95] Jan Reineke, Isaac Liu, Hiren D Patel, Sungjun Kim, and Edward A Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ ISSS)*, 2011.

[96] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Appendix to drambulism: Balancing performance and predictability through dynamic pipelining. http://hdl.handle.net/10012/15678, 2020.

[97] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. Memory access scheduling. *ACM SIGARCH Computer Architecture News*, 2000.

[98] Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. Towards time-predictable data caches for chip-multiprocessors. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 180–191. Springer, 2009.

[99] Ioannis Schoinas, Babak Falsafi, Alvin R Lebeck, Steven K Reinhardt, James R Larus, and David A Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 297–306, 1994.

[100] Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. Modeling cache coherence to expose interference (artifact). In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[101] Nathanal Sensfelder, Julien Brunel, and Claire Pagetti. On how to identify cache coherence: Case of the nxp qoriq t4240. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2020.

[102] Ashok Singhal, Bjorn Liencres, Jeff Price, Frederick M Cerauskis, David Broniarczyk, Gerald Cheung, Erik Hagersten, and Nalini Agarwal. Implementing snooping on a split-transaction computer system bus, November 2 1999. US Patent 5,978,874.

[103] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis lectures on computer architecture*, 6(3):1–212, 2011.

[104] Nivedita Sritharan, Anirudh Kaushik, Mohamed Hassan, and Hiren Patel. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 433–445. IEEE, 2019.

[105] Nivedita Sritharan, Anirudh M Kaushik, Mohamed Hassan, and Hiren Patel. Hourglass: Predictable time-based cache coherence protocol for dual-critical multi-core systems. *arXiv preprint arXiv:1706.07568*, 2017.

[106] DDR3 SDRAM Standard. Jedec jesd79-3, 2007.

[107] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. Bliss: Balancing performance, fairness and complexity in memory access scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):3071–3087, 2016.

[108] Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*, pages 300–303, 2008.

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

[110] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[111] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 239–243. IEEE, 2007.

[112] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. Omaking shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 157–167. IEEE, 2013.

[113] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[114] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009.

[115] Zheng Pei Wu, Yo Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 372–383. IEEE, 2013.

[116] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 372–383. IEEE, 2013.

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

[118] Zhuanhao Wu, Anirudh Mohan Kaushik, Paulos Tegegn, and Hiren Patel. A hardware platform for exploring predictable cache coherence protocols for real-time multicores. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 92–104. IEEE, 2021.

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

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

[121] Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[122] Wei Zhao and John A Stankovic. Performance analysis of fcfs and improved fcfs scheduling algorithms for dynamic real-time computer systems. In *[1989] Proceedings. Real-Time Systems Symposium*, pages 156–165. IEEE, 1989.

[123] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 1–6. IEEE, 2010.