Predictable Shared Memory Resources for Multi-Core Real-Time Systems

by

Mohamed Hassan

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, 2017

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	Tulika Mitra Professor
Supervisor	Hiren Patel Associate Professor
Internal Member	Rodolfo Pellizzoni Associate Professor
Internal Member	Catherine Gebotys Professor
Internal-external Member	Bernard Wong Associate Professor

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

In what follows is a list of publications which I have co-authored and used their content in this dissertation. For each publication, I present a list of my contributions.

The use of the content, from the listed publications, in this dissertation has been approved by all co-authors.

1. Mohamed Hassan, Hiren Patel, "Requirement- and Criticality-aware Bus Arbitration for Mixed Criticality Systems" *In IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016), Vienna, Austria* [1]

    - Developed the arbitration algorithm and implemented it
    - Conducted the timing analysis
    - Designed and executed the experiments
    - Analyzed the experimental results
    - Wrote a significant portion of the article

2. Mohamed Hassan, Anirudh M. Kaushik, Hiren Patel, "A Predictable Cache Coherence for Multi-core Real-time Systems" *In IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2017), Pittsburgh, PA, USA* [2]

    - Identified the sources of unpredictability in conventional coherence protocols
    - Defined the invariants that maintains predictable behaviors
    - Developed the predictable coherence protocol (PMSI)
    - Formulated the optimization problem
    - Conducted the timing analysis
    - Involved in designing the experiments
    - Involved in analyzing the experimental results
    - Wrote a significant portion of the article

3. Mohamed Hassan, Hiren Patel, Rodolfo Pellizzoni, "A Framework for Scheduling DRAM Memory Accesses for Multi-Core Mixed-time Critical Systems" *In IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2015), Seattle, USA* [3]

- Developed the mixed page policy
- Developed the optimal harmonic TDM schedule
- Formulated the optimization problem
- Conducted the timing analysis
- Designed and executed the experiments
- Analyzed the experimental results
- Wrote a significant portion of the article

4. Mohamed Hassan, Hiren Patel, Rodolfo Pellizzoni, " PMC: A Requirement-aware DRAM Controller for Multi-core Mixed Criticality Systems" *In ACM Transactions on Embedded Computing Systems (TECS)* [4]

    - Extended PMC to support multi-rank DRAMs
    - Extended PMC to support smaller transaction sizes
    - Designed and executed the experiments
    - Analyzed the experimental results
    - Wrote a significant portion of the article

5. Mohamed Hassan, Anirudh M. Kaushik, Hiren Patel, "Reverse Engineering Embedded Memory Controllers through Latency-based Analysis" *In IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2015), Seattle, USA* [5]

    - Developed the reverse-engineering algorithms
    - Defined the inference rules used in the reverse-engineering process
    - Conducted the latency-based analysis
    - Involved in designing the experiments
    - Involved in analyzing the experimental results
    - Wrote a significant portion of the article

6. Mohamed Hassan, Anirudh M. Kaushik, Hiren Patel, " Exposing Implementation Details of DRAM Memory Controllers through Latency-based Analysis" *In IEEE Transactions on Computers (TC) (Under Review)* [6]

    - Added the reverse-engineering of the write buffer policy

- Extended the reverse-engineering of the XOR address mapping
- Identified potential applications for exposing the memory controller details
- Involved in designing the experiments for the FPGA
- Involved in analyzing the experimental results from the FPGA
- Wrote a significant portion of the article

7. Mohamed Hassan, Hiren Patel, "MCXplore: An Automated Framework for Validating Memory Controller Designs" *In IEEE Design, Automation and Test Conference in Europe (DATE 2016), Dresden, Germany* [7]

   - Defined the formal models of the input stimulus of the memory system
   - Implemented these models in a modeler checker tool and used them as a test generation engine
   - Developed a methodology to use DRAM utilization to validate the memory controller behavior
   - Implemented the MCXplore framework as an open-source tool
   - Developed the regression test suites that stress the DRAM behavior
   - Showed case studies on using MCXplore to validate state-of-the-art DRAM policies
   - Applied the proposed methodology to discover timing violations in memory controllers
   - Wrote a significant portion of the article

8. Mohamed Hassan, Hiren Patel, " MCXplore: Automating the Validation Process of DRAM Memory Controller Designs" *In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)(Under Review)* [8]

   - Illustrated the usage of MCXplore with other high-level statistics
   - Extended the framework to validate other DRAM policies
   - Designed and executed the experiments
   - Analyzed the experimental results
   - Wrote a significant portion of the article

Abstract

A major challenge in multi-core real-time systems is the interference problem on the shared hardware components amongst cores. Examples of these shared components include buses, on-chip caches, and off-chip dynamic random access memories (DRAMs). The problem arises because different cores in the system interfere with each other, while competing to access the shared hardware components. It is a challenging problem for real-time systems because operations of one core affect the temporal behaviour of other cores, which complicates the timing analysis of the system. We address this problem by making the following contributions. 1) ***For shared buses***, we propose CArb, a predictable and criticality-aware arbiter, which provides guaranteed and differential service to tasks based on their requirements. In addition, we utilize CArb to mitigate overheads resulting from system switching among different modes. 2) ***For the cache hierarchy***, we address the problem of maintaining cache coherence in multi-core real-time systems by modifying current coherence protocols such that data sharing is viable for real-time systems in a manner amenable for timing analysis. The proposed solution provides performance improvements, does not impose any scheduling restrictions, and does not require any source-code modifications. 3) ***At the shared DRAM level***, we propose PMC, a programmable memory controller that provides latency guarantees for running tasks upon accessing the off-chip DRAM, while assigning differential memory services to tasks based on their bandwidth and latency requirements. In addition to PMC, we conduct a latency-based analysis on DRAM memory controllers (MCs). Our analysis provides both best-case and worst-case bounds on the latency that any request suffers upon accessing the DRAM. The analysis comprehensively covers all possible interactions of successive requests considering all possible DRAM states. Finally, we formally model request interrelations and DRAM command interactions. We use these models to develop an automated validation framework along with benchmark suites to validate and evaluate PMC and any other MC, which we release as an open-source tool.

Acknowledgements

First and foremost, all the praise and thankfulness to Allah for empowering me to complete this thesis. Without his help, I would not have the ability to pass the finish line of this stage in my life.

I would like to seize this opportunity to thank Professor Hiren Patel (my supervisor), and Professor Rodolfo Pellizzoni for their valuable help and guidance. I would also like to thank my committee members: Professor Tulika Mitra, Professor Catherine Gebotys, Professor Rodolfo Pellizzoni, and Professor Bernard Wong for taking the time and effort to participate in my examination committee and provide me with valuable feedback.

I would like to thank my friends in the Computer Architecture and Embedded Systems Research (CAESR) group at the University of Waterloo for all the great collaborations, discussions, and projects: Anirudh M. Kaushik, Nivedita Sritharan, Yunling Cui, Zhuoran Yin, and Danlu Guo.

I cannot find the words to thank my parents for their continuous and endless support throughout my whole life. I thank them for believing in me and for helping me be who I am today. Thanks to my brother, Alaa, my sister, Shimaa, and my dear friend, Youssef Elkady for the unconditional love and support.

I would like to express my deep gratitude to my wife, Hebatullah. Thanks for her patience, support, love, sacrifices, and consideration. I also thank my son, Albaraa, and my daughter, Jannah, for shining our home with their beautiful smiles.

Dedication

Indeed, my prayer, my rites of sacrifice, my living and my dying are for Allah , Lord of the worlds. [Qura'n 6:162]

Table of Contents

List of Tables xvi

List of Figures xviii

List of Abbreviations xxii

List of Symbols xxv

1 Introduction 1

2 Criticality- and Requirement-aware Bus Arbitration for Multi-core Mixed Criticality Systems 4

2.1 Introduction . . . 4

2.1.1 Contributions . . . 6

2.2 Related Work . . . 7

2.3 System Model . . . 8

2.4 Execution Time Decomposition . . . 10

2.4.1 Illustrative Example . . . 10

2.5 Applicability of real-time arbiters in MCS . . . 11

2.6 CArb: Proposed Arbitration Scheme . . . 13

2.6.1 Inter-class Arbitration . . . 14

2.6.2 Intra-class Arbitration . . . 15

2.6.3 Area Overhead . . . 17
2.7 WC Analysis and Problem Formulation . . . 17
2.7.1 WC analysis . . . 18
2.7.2 Optimization problem formulation . . . 19
2.7.3 $\Lambda(L)$: The WC number of memory accesses as a function of CL . . . 21
2.8 Dynamic Re-arbitration . . . 21
2.8.1 Motivation . . . 21
2.8.2 Proposed Solutions . . . 21
2.8.3 Effect of Re-arbitration on Lower-criticality Tasks . . . 24
2.9 Experimental Evaluation . . . 25
2.9.1 Avionics Use-case . . . 26
2.9.2 Minimum Achievable Latency . . . 27
2.9.3 Synthetic Experiments . . . 27
2.9.4 Dynamic Re-arbitration . . . 29
2.10 Summary . . . 32

3 PMSI: Predictable Cache Coherence for Multi-core Real-time Systems 33
3.1 Introduction . . . 33
3.1.1 Contributions . . . 35
3.2 Related Work . . . 35
3.3 Background: Cache Coherence . . . 37
3.3.1 Transient Cache Coherence States . . . 39
3.4 System Model . . . 40
3.5 Sources of Unpredictability Due to Coherence . . . 41
3.5.1 Source *1*: Inter-core Coherence Interference on Same Line . . . 42
3.5.2 Source *2*: Inter-core Coherence Interference on Different Lines . . . 43
3.5.3 Source *3*: Inter-core Interference Due to Write Hits . . . 44
3.5.4 Source *4*: Intra-core Coherence Interference . . . 46

3.6 PMSI: A Predictable Coherence Protocol . . . 47
3.6.1 Architectural Modifications . . . 48
3.6.2 Coherence Protocol Modifications . . . 49
3.7 Latency Analysis . . . 52
3.8 Evaluation . . . 56
3.8.1 Verification . . . 57
3.8.2 Exp.1: Bounding the Memory Latency . . . 57
3.8.3 Exp.2: Comparing Performance with Conventional Protocols and Alternative Predictable Approaches . . . 59
3.8.4 Exp.3: Comparing to the Ideal Scenario . . . 60
3.8.5 Exp.4: Scalability . . . 61
3.9 Summary . . . 62

4 PMC: A Requirement-aware DRAM Controller for Multi-core Mixed Criticality Systems 63

4.1 Introduction . . . 63
4.1.1 Contributions . . . 64
4.2 Background: Main Memory . . . 66
4.2.1 Memory Page Policies . . . 68
4.3 Related Work . . . 69
4.4 Arbitration Mechanisms . . . 71
4.5 PMC: The Proposed Solution . . . 72
4.5.1 PMC Architecture . . . 72
4.5.2 Formulating Bundles . . . 74
4.5.3 Dynamic Bank Interleaving . . . 76
4.5.4 Rank Interleaving . . . 77
4.5.5 Arbitration Logic . . . 78
4.6 Schedule Generation . . . 79

4.6.1 Proposed Implementation . . . 80
4.6.2 Schedule parameters . . . 82
4.6.3 Schedule Slots . . . 82
4.7 Timing Analysis . . . 84
4.7.1 Problem Formulation . . . 86
4.8 Experimental Evaluation . . . 88
4.8.1 Case-study: Multimedia System . . . 88
4.8.2 Synthetic Experiments . . . 90
4.9 Summary . . . 98

5 Reverse Engineering Embedded DRAM Memory Controllers through Latency-based Analysis 99
5.1 Introduction . . . 99
5.1.1 Contributions . . . 100
5.2 Related Work . . . 101
5.3 Memory Latency Analysis . . . 102
5.3.1 Proof Strategy . . . 103
5.3.2 Example 1: Two accesses with same access type to two different banks in the same rank . . . 105
5.3.3 Example 2: Two accesses with different access type to two different banks in the same rank . . . 106
5.4 Reverse-Engineering Properties of the MC . . . 108
5.4.1 Reverse-engineering page policy and address mapping . . . 108
5.4.2 Reverse-engineering the command arbitration scheme . . . 114
5.4.3 Advanced MC features . . . 115
5.4.4 Performance Counters . . . 117
5.5 Potential Applications . . . 117
5.6 Experimental Evaluation . . . 119
5.6.1 Reverse-engineering MC's properties of the XUPV5-LX110T platform . 119
5.6.2 Evaluation on simulation framework . . . 122
5.7 Summary . . . 127

6 MCXplore: Automating the Validation Process of DRAM Memory Controller Designs 128

6.1 Introduction . . . 128
6.1.1 Contributions . . . 129
6.2 Background: Model Checking . . . 130
6.3 Related Work . . . 131
6.3.1 Formal Verification of the Memory System . . . 132
6.4 MCXplore Methodology . . . 134
6.4.1 Proposed Models . . . 135
6.5 Validating MC's Frontend . . . 138
6.5.1 Address Mapping Policies . . . 139
6.5.2 Page Management Policies . . . 147
6.5.3 Arbitration Schemes . . . 149
6.6 Validating MC's backend . . . 151
6.6.1 tCCD . . . 152
6.6.2 tRC . . . 154
6.6.3 tFAW . . . 155
6.6.4 tRTRS . . . 156
6.6.5 tRTP . . . 158
6.6.6 tRCD,tWL, and tRL . . . 159
6.6.7 tRRD . . . 161
6.6.8 tWR . . . 163
6.6.9 tWTR . . . 164
6.6.10 Summary . . . 165
6.6.11 Smart Refresh . . . 167
6.6.12 Command Bus Contention . . . 167
6.7 Extensibility of MCXplore . . . 169
6.8 Summary . . . 170

7 Bounding Total Memory Latency in Multi-Core Real-Time Systems **171**

7.1 System Model 171

7.2 Timing Analysis of Coherence Interference Assuming CArb's Schedule 172

7.3 Aggregated Memory Latency Analysis 174

8 Conclusion and Future Work **176**

References **178**

List of Tables

2.1 Experiment using the avionics use-case from Honeywell [9]. 25

2.2 Parameters of synthetic experiments. 28

2.3 Parameters of the dynamic case experiment. 30

3.1 Transient states between S and I in a conventional MSI protocol. *issue msg/state* means the core issues the message *msg* and move to state *state*. *-/state* indicates that there is no *msg* issued. Shaded cells represent the situations where no transition occurs, while cells marked with X denote impossible cases under correct operation [10]. 40

3.2 Private memory states for PMSI. *issue msg/state* means the core issues the message *msg* and move to state *state*. A core issues a *load/store* request. Once the cache line is available, the core *reads/writes* it. A core needs to issue a *replacement* to write back a dirty block before eviction. Changes to conventional MSI are in bold red. 49

3.3 Semantics of the proposed transient states to achieve a predictable behavior. . . . 52

4.1 Important JEDEC timing constraints (DDR3-1333) [11]. 68

4.2 Different bank interleaving for a single rank DDR3-1333. Bundle widths are in cycles. 77

4.3 Terms and brief descriptions. 83

4.4 Multimedia processing system requirements. 89

5.1 Best and worst-case latencies. 107

5.2 Results of XORing different bank and row bits. 115

5.3 DDR2 specifications. 120
5.4 Address mapping findings for XUPV5-LX110T. 121
5.5 System configuration. 122
5.6 MC configurations. 123

6.1 Currently supported configurations. 136
6.2 Validating $tRTP$. 159
6.3 Validating $tRCD$, tRL and tWL. 161
6.4 Tests of timing parameters. 166

List of Figures

1.1 Multi-core architecture. 2

2.1 Multi-core architecture. 8
2.2 Real-time arbiters. 12
2.3 Memory bus arbitration using CArb. 14
2.4 Avionics use-case results. 26
2.5 Latency requirements. 28
2.6 Synthetic experiments (y-axis is the total WCL, y^{tot}). 29
2.7 Effect of decreasing core frequency on tasks of C_2. 31

3.1 Cache coherence. 38
3.2 Unpredictability source 1: inter-core coherence interference on same line. Initially, c_0 modified A. c_2 is under analysis. 43
3.3 Unpredictability source 2: inter-core coherence interference on different lines. Initially, c_0 modified A and B. c_1 is under analysis. 44
3.4 Unpredictability source 3: inter-core interference due to write hits. 45
3.5 Unpredictability source 4: intra-core coherence interference. Initially, c_0 has modified A. c_2 is under analysis. 47
3.6 Architectural changes necessary for PMSI. 48
3.7 Transient states example; grey boxes are events, and arrows are state transitions. Initially, c_0 has A in S. 51
3.8 Different latency components. Initially, c_0 modified B and c_1 modified A. 53

3.9 The latency bound on each interference component. Empty slots/periods do not have events that are related to c_1's latency. 55
3.10 WC latencies and the effect of unpredictability sources on them. Unpredictable i corresponds to source i in Section 3.5. Horizontal dotted line represents the analytical bound. 58
3.11 Execution time slowdown compared to MESI protocol. 60
3.12 Slowdown in the execution time of different approaches compared to ideal scenario. 61
3.13 Latencies with different number of cores. 62

4.1 DRAM subsystem. 66
4.2 A write access followed by a write or read access targeting the same bank and rank for close-page policy. 67
4.3 PMC framework. 73
4.4 Command arrangements of the four bundles interleaving across 8 banks of DDR3-1333 for a write request. A: ACTIVATE command, C: CAS command, CP: CASp command, and N: NOP command. 74
4.5 Bundles usage for the 4 bank interleaving case. 75
4.6 A write followed by a read both targeting an open row. 78
4.7 Command arrangements of the four bundles interleaving across 2 ranks and 4 banks per rank. 79
4.8 A write followed by a read for no bank interleaving (single bank bundle). 80
4.9 TDM scheduling mechanisms. 81
4.10 A schedule example. 84
4.11 Results for the multimedia processing system use-case. 89
4.12 Effect of varying $kmax$ and s of SRT requestors. 91
4.13 Effect of varying number of requestors. 93
4.14 Effect of the transaction size. 94
4.15 Dynamic *vs.* static bank interleaving. 95
4.16 Effect of rank interleaving on memory latency. 97

5.1 A write access followed by a write or read access targeting the same bank and rank. A: ACTIVATE, W: WRITE, R: READ, P: PRE, D: DATA. 103
5.2 The two conditions controlling the issuance of the first command of pr_2. C1 and C2 represent CAS1 and CAS2 commands respectively. 104
5.3 Two accesses with same access type to two different banks in the same rank. . . . 105
5.4 A read access followed by a write access targeting different banks in the same rank. 106
5.5 Reverse-engineering process. 109
5.6 Latency bounds for a sequence of two requests. 110
5.7 Latency plots for $test_1$ stimulating the on-board MC of XUPV5-LX110T. OP: open-page policy, and CP: close-page policy. 120
5.8 Latency plots for page policy and address mapping inference tests. 123
5.9 Latency plot for hybrid-page policy. 124
5.10 Latency plot for FR-FCFS threshold test. 126
5.11 Write buffer policy. 127

6.1 Model checking operation. 131
6.2 Proposed validation process of MCs. 134
6.3 DRAM commands and timing constraints interaction. Subscripts reflect the targeted bank and rank, respectively. d: different, s: same, x: do not care. Di: start of the data transfer. De: end of the data transfer. P is for same bank. $A_{d,s}$ is an A command to a different bank on the same rank. 138
6.4 Test generation for validating XOR mapping 140
6.5 XOR address mapping. 141
6.6 Command sequence of $Suite_{XOR}$ on XOR mapping. 141
6.7 Address masking operation. 143
6.8 Address masking schemes. 143
6.9 Address masking results. 144
6.10 Command sequence from executing $Test_i$ on $Scheme_i$, $i \in \{1, 2, 3\}$. 144
6.11 Rank hopping. 146

6.12 Evaluation of page policies. . . . 148
6.13 Command arrangement for $Test_{PP}$. . . . 149
6.14 Evaluation of FR-FCFS threshold. . . . 150
6.15 Command sequence of $Suite_{thr}$ when $hit = thr - 1$. . . . 151
6.16 Validation dependency graph for timing parameters. . . . 152
6.17 Command sequence of $Test_{CCD}$. . . . 153
6.18 Test generation for validating $tCCD$. . . . 153
6.19 $tCCD$ validation results. . . . 154
6.20 tRC validation results. . . . 155
6.21 $tFAW$ validation results. . . . 156
6.22 $tRTRS$ validation results. . . . 157
6.23 Command sequence of $Test_{RTRS}$. . . . 158
6.24 Command sequence of $Test_{RTP}$. . . . 159
6.25 $tRTP$ validation results. . . . 159
6.26 Validating $tRCD$, tRL, and tWL. . . . 160
6.27 $tRCD$, tRL, and tWL validation results. . . . 161
6.28 $tRRD$ validation results. . . . 162
6.29 Command sequence of $Test_{RRD}$. . . . 163
6.30 tWR validation results. . . . 164
6.31 Command sequence of $Test_{WR}$. . . . 164
6.32 $tWTR$ validation results. . . . 166
6.33 Command sequence of $Test_{WTR}$. . . . 166
6.34 Smart Refresh behaviour with different number of accessed rows. . . . 168
6.35 Policies to resolve command bus contention. . . . 169

7.1 An architectural example for the system model used in deriving the total WC memory aggregated latency. The system has three cores and three tasks, where τ_{11} is mapped to c_0, τ_{12} is mapped to c_1, and τ_{13} is mapped to c_2. . . . 173

List of Abbreviations

ABS	Anti-lock Brake System
AMC	Analyzable Memory Controller
BMC	Bounded Model Checking
BW	Bandwidth
CCSP	Credit Control Static Priority
CL	Criticality Level
COP	conservative Open Page
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
DIMM	Dual In-line Memory Modules
DMA	direct Memory Access
DoS	Denial of Service
DRAM	Dynamic Random Access Memories
FCFS	First-Come First-Serve
FIFO	First-In First-Out
FR-FCFS	First-Ready First-come First-Serve
FSM	Finite State Machine
GPU	Graphical Processing Unit
HRR	Harmonic Round Robin
HRT	Hard Real-Time
HWRR	Harmonic Weighted Round Robin

IBB	Initial Bank Bits
LLC	last-level cache
LRU	Least Recently Used
LRU	Pseudo Least Recently Used
LTL	Linear Temporal Logic
LUT	Look Up Table
MC	Memory Controller
MCS	Mixed Criticality Systems
MIG	Memory Interface Generator
MSI	Modify–exclusive-Share-Invalidate coherence protocol
MSI	Modify-Share-Invalidate coherence protocol
PMC	Programmable Memory Controller
PMSI	Predictable MSI coherence protocol
PMU	Performance Monitoring Unit
PR	Pending Request
PRR	Prioritized Round Robin
PWB	Pending Writeback
RM	Rate Monotonic
RR	Round Robin
RTL	Register Transfer Level
SRT	Soft Real-Time
SVA	system Verilog Assertion
TDM	Time Division Multiplexing
TLM	Transaction Level Modeling
WC	Worst Case
WCET	Worst-Case Execution Time

WCL Worst Case Latency
WRR Weighted Round Robin

List of Symbols

C_l	Criticality class l
D_{jl}	Deadline of task τ_{jl}
E_{jl}	Total WCET. $E_{jl} = S_{jl} + I_{jl}$.
Γ_l	total number of tasks in C_l
H	Hyperperiod of a CArb schedule
$I_{jl}(L)$	WC additional latency due to inter-task interference. S_{jl} is a function of L.
L_l	Criticality level l
$S_{jl}(L)$	WCET of any job of task τ_{jl} when it runs in isolation (no inter-task interference). S_{jl} is a function of L.
τ_{jl}	Task j in criticality class l
T_{jl}	Minimum inter-arrival time for jobs of task τ_{jl}, which represents the task period
CH_l	The hyperperiod of class C_l in a CArb schedule
CP_l	The class period of C_l in a CArb schedule
CW_l	The class weight of C_l in a CArb schedule

γ_r	Total number of tasks in partition r for partitioned scheduling.
Λ_{jl}	maximum number of memory accesses issued by any job of τ_{jl}.
M_{jl}	Memory latency requirement of task τ_{jl}.
$\mathcal{T}P_{jl}$	The task period of τ_{jl} in a CArb schedule
$\mathcal{T}W_{jl}$	The task weight of τ_{jl} in a CArb schedule
$\hat{U}_r$	The partial utilization granted to partition r for partitioned scheduling
U_r	Total utilization of tasks in partition r for partitioned scheduling.
y^{acc}	CArb's access latency of a request req_r generated by τ_{jl}.
$y^{sch}_{jl,r}$	CArb's scheduling latency of a request req_r generated by τ_{jl}
y^{tot}_{jl}	CArb's total worst-case latency of any request generated by τ_{jl}
Z_l	The window size of a CArb schedule, which represents the number of task slots in a class slot
BWL_i	The minimum bandwidth required by r_i
LR_i	The memory access latency requirement of r_i
R	The set of requestors in the system deploying PMC
$kmax_i$	Maximum number of bundles of a requestor r_i that are serviced per sub-request

LBB_i	The lower-bound bandwidth delivered to a requestor r_i
p_i	Harmonic period: the interval (in slots) between two successive executions of i. It is equal to the total number of slots divided by s_i
pr_i	r_i's relative priority
p_i	Harmonic period: the interval (in slots) between two successive executions of i. It is equal to the total number of slots divided by s_i
r_i	Requestor number i in the system deploying PMC
s_i	Harmonic slots: total number of slots allocated to requestor r_i
$tBUS$	Time required to transfer a data burst on the data bus
$tFAW$	The number of cycles in which four activates are allowed within the same rank
$tRCD$	The minimum time between activating the row and a read/write access to it
tRC	The minimum time between two accesses to different rows in a bank
tRL	The minimum time between a read CAS and the start of data transfer
tRP	The row pre-charge time
$tRTRS$	The rank to rank switching delay
$tRTW$	Read to write switching delay
tWL	The minimum time between a write CAS and the start of data transfer

$tWTR$	Write to read switching delay
UBD_i	The upper-bound latency incurred by a memory request from r_i
w_i	The width of slot j in clock cycles
Y_j	The total number of requestors assigned to slot j
L^{acc}	The time required to transfer the requested data by c_i between the shared memory and the private cache of c_i
L_{ir}^{arb}	Arbitration latency of request, $req_{i,r}$, generated by core c_i
WCL_i^{arb}	The worst-case arbitration latency of any request generated by core c_i
$L_{i,r}^{coh}$	Coherence latency of request, $req_{i,r}$, generated by core c_i
$WCL_i^{interCoh}$	The worst-case inter-coherence latency of any request generated by core c_i
$L_{i,r}^{interCoh}$	Inter-core coherence latency of request, $req_{i,r}$, generated by core c_i
$WCL_i^{intraCoh}$	The worst-case intra-coherence latency of any request generated by core c_i
$L_{i,r}^{intraCoh}$	Intra-core coherence latency of request, $req_{i,r}$, generated by core c_i
t_i	The arrival time of the first command from pr_i to the command queue
l_i^{BEST}	The best-case latency of pr_i
BKW	The number of DRAM bank bits

bnk_i	The target bank of the i^{th} physical request
ch_i	The target channel of the i^{th} physical request
CLW	The number of DRAM column bits
CNW	The number of DRAM channel bits
cl_i	The target column of the i^{th} physical request
cs_i	The memory commands sequence of the i^{th} physical memory request
f_i	The time-stamp at which pr_i starts its data transfer
l_i	The latency of pr_i defined as $l_i = f_i - t_i$
la_i	The address of the i^{th} logical memory request
lr_i	The i^{th} logical memory request
o_i	The type of the i^{th} logical memory request. It is either a read or a write, $o \in \{R, W\}$
pa_i	The address of the i^{th} physical memory request
pr_i	The i^{th} physical memory request
PW	The number of bits in the physical address
RKW	The number of DRAM rank bits
rnk_i	The target rank of the i^{th} physical request

RWW	The number of DRAM row bits
rw_i	The target row of the i^{th} physical request
l_i^{WORST}	The worst-case latency of pr_i
$L1_i^{Rhits}$	The number of read requests issued by core c_i and are hits in L1 cache
$L1_i^{Whits}$	The number of write requests issued by core c_i and are hits in L1 cache
$L1^{accL}$	L1 hit latency specified by the processor datasheet
$L2_i^{hits}$	The number of requests issued by core c_i and are hits in L2 cache
$L2_i^{misses}$	The number of requests issued by core c_i and are misses in L2 cache
MAL_i	The total WC memory aggregated latency of a core c_i
S^{L2}	The slot width of the TDM schedule managing accesses to L2 cache
WCL_i^{DRAM}	The WCL of a read or write request issued by core c_i upon accessing DRAM
WCL_i^{RL1}	The WCL of a read request issued by core c_i upon accessing L1 cache
WCL_i^{WL1}	The WCL of a write request issued by core c_i upon accessing L1 cache
WCL_i^{L2}	The WCL of a read or write request issued by core c_i upon accessing L2 cache

Chapter 1

Introduction

Real-time systems are those, whose behaviour depends not only on their functionality, but also on their response time. Until recently, real-time systems have been limited to safety-critical domains such as avionics and spacecrafts. However, with the emanating cyber-physical systems (CPS) and Internet of Things (IoT) revolution, real-time systems are becoming ubiquitous in many emerging domains [12]. Examples include transportation (such as automobiles, avionics, space vehicles, etc.), infrastructures such as power grids, health care (such as medical devices, and implantable and biomedical devices), and industrial environment (such as manufacturing plants, power plants, and robots) to name a few.

Since application demands from these emerging domains continue to increase, there is a surge in wanting to use multi-core platforms to deploy real-time systems. This is primarily due to the benefits multi-core platforms provide in cost, and performance. However, multi-core platforms impose new challenges towards guaranteeing requirements of running applications. These challenges arise because different cores in the platform may interfere each other, while competing to access memory resources that are shared amongst them. As Figure 1.1 depicts, these shared resources include buses, on-chip caches, and off-chip dynamic random access memories (DRAMs). This interference is a challenge for real-time systems because operations of one core affect the temporal behaviour of other cores, which complicates the timing analysis of the system. Timing analysis is crucial for real-time systems to guarantee that tasks meet their timing requirements such that the task's worst-case execution time (WCET) is less than its deadline. Hence, the highlighted interference challenge is the focus of this thesis. In particular, this thesis proposes predictable architectures for shared buses, caches, and DRAMs. We believe that providing predictable solutions for shared memory resources is of unavoidable necessity towards deploying real-time systems onto multi-core platforms. In addition, the problems associated with this interference are exacerbated in real-time systems deploying tasks with different criticalites,

known as *mixed criticality systems* (MCS) since a non-critical task can affect the behaviour of a critical task. Consequently, this thesis pays special attention to MCS.

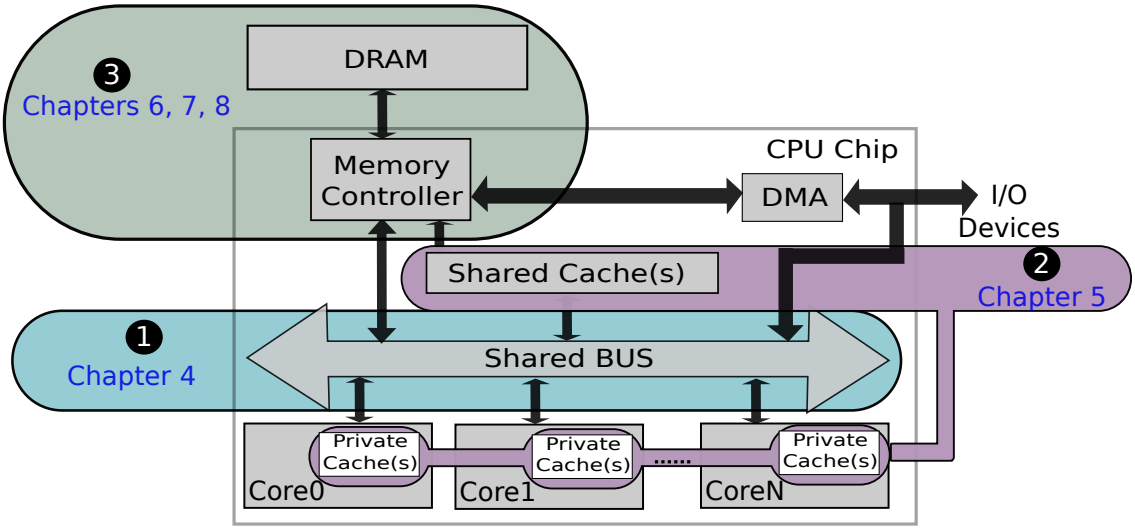


Figure 1.1: Multi-core architecture.

We address the interference problem on the ***shared buses*** (❶ in Figure 1.1) in multi-core MCS by proposing CArb [1], a predictable and criticality-aware arbiter (Chapter 2). In addition, we utilize CArb to mitigate overheads resulting from system switching among different modes, which is one of the onerous challenges in MCS. CArb does not impose any restrictions on mapping applications to cores. Hence, it operates in tandem with existing operating system (OS) scheduling policies. CArb is able to dynamically adapt arbitration at run time to respond to increases in the monitored execution times of tasks. Utilizing this adaptation, CArb is able to offset these increases; hence, it postpones the OS's need to switch to a degraded mode. Our evaluation using an avionics case study from Honeywell shows that CArb is able to meet diverse requirements of tasks with mixed criticalities.

At the ***cache hierarchy*** level (❷ in Figure 1.1), one of the challenging burdens for computer architects is to maintain correctness of shared data stored in cache hierarchies of multi-core platforms, which is known as cache coherence [10]. Although cache coherence has been extensively investigated for conventional performance-oriented platforms, multi-core real-time systems introduce new challenges from the predictability perspective. To exemplify, we show that deploying a conventional coherence protocol in a predictable system can lead to unpredictable

behaviours [2]. In Chapter 3, we highlight those challenges and provide a set of invariants to address them. These invariants are architecture- and protocol-independent. Based on these invariants, we propose the first predictable hardware cache coherence protocol, named PMSI, which enables tasks to simultaneously share data in a manner amenable for timing analysis. PMSI augments the classic modify-share-invalid (MSI) protocol with transient coherence states, and minimal architectural changes. This allows us to derive worst-case (WC) latency bounds that provide timing guarantees. Our empirical results show that PMSI improves average-case performance by up to $4\times$ over the next best alternative, which avoids caching the shared data on the private cache of each core.

At the ***shared DRAM*** level (❸ in Figure 1.1), we propose PMC [3, 4], a programmable memory controller that provides latency guarantees for running tasks upon accessing the off-chip DRAM, while assigning differential memory services to tasks based on their bandwidth and latency requirements (Chapter 4). PMC supports MCS by enabling the system designer to specify requirements per software task. Leveraging awareness of these requirements, PMC optimally assigns differential memory service per task. We also presented a formal timing analysis that proves latency and bandwidth guarantees.

In addition to PMC, we conduct a latency-based analysis on DRAM memory controllers (MCs) [5,6]. Our analysis provides both best-case and worst-case bounds on the latency that any request suffers upon accessing the DRAM (Chapter 5). The analysis comprehensively covers all possible interactions of successive requests considering all possible DRAM states. To show the effectiveness of this analysis, we use it to reverse-engineer properties deployed in state-of-the-art embedded memory controllers. We execute carefully-crafted yet simple C programs and measure the memory latency through performance counters. Afterwards, we use the proposed analysis to infer certain properties of the memory controller.

Finally, to validate and test the performance of our work at the DRAM level, we present an automated framework for the validation of DRAM MCs, called MCXplore [7, 8] (Chapter 6). In developing MCXplore, we construct formal models for memory requests interrelation and DRAM command interaction. The framework enables validation engineers to define their test plans precisely as temporal logic specifications. We use the NuSMV model-checker to generate counter-examples that serve as test templates; hence, MCXplore uses these test templates to generate memory tests to validate the correctness properties of the memory controller. We show the effectiveness of MCXplore by validating various state-of-the art MC features as well as hard-to-detect timing violations that often occur. We also provide a set of predefined test plans, and regression tests that validate essential properties of modern DRAM MCs. We release MCXplore as an open-source framework to allow validation engineers and researchers to extend and use.

Chapter 2

Criticality- and Requirement-aware Bus Arbitration for Multi-core Mixed Criticality Systems

This chapter presents CArb, an arbiter for controlling accesses to the shared memory bus in multi-core mixed criticality systems. CArb is a requirement-aware arbiter that optimally allocates service to tasks based on their requirements. It is also criticality-aware since it incorporates criticality as a first-class principle in arbitration decisions. CArb supports any number of criticality levels and does not impose any restrictions on mapping tasks to cores. Hence, it operates in tandem with existing scheduling policies. In addition, CArb is able to dynamically adapt memory bus arbitration at run time to respond to increases in the monitored execution times of tasks. Utilizing this adaptation, CArb is able to offset these increases; hence, postpones the system need to switch to a degraded mode. We prototype CArb, and evaluate it with an avionics case-study from Honeywell as well as synthetic experiments.

2.1 Introduction

MCS consist of a set of interacting software components, where the components may operate under various criticality levels (CLs) [9]. Each CL provides a degree of assurance against the software component's failure [13]. For instance, DO178C used in avionics denotes five CLs

ranging from critical to no effect. The real-time research community is interested in using multi-cores to deploy MCS, mainly because multi-cores offer small-sized, low-weighted, and low-cost hardware platforms that are mainstream nowadays. However, this requires consolidating software components onto the multi-core platform, which implies sharing hardware resources such as buses, caches and main memories among these components. Resource sharing brings out a key challenge in the design of MCS: to effectively schedule shared hardware resources so as to ensure safety guarantees mandated by the CLs, and to deliver the performance demanded by each software component. Recent efforts addressing this challenge have focused on proposing models and scheduling algorithms that schedule tasks with CLs onto cores [14–18]. Earlier approaches proposed methods to deploy MCS onto single core platforms [14, 15], which were further advanced to multi-core platforms [16–18].

These efforts developed a standard model for MCS, where each task is characterized by a criticality level, usually two CLs: LO and HI. Each task has WCET estimate, S, for each CL, $S(LO)$ and $S(HI)$ for the two levels case. The system operates initially in a normal mode, where it considers the $S(LO)$ of each task and both higher- and lower-critical tasks utilize the hardware resources. If a critical task exceeds its $S(LO)$, the system switches to a degraded mode, where it suspends all lower-critical tasks and considers the $S(HI)$ of the higher-critical ones [19]. These dynamic migrations between various modes is a key characteristic of MCS as compared to single-criticality traditional real-time systems. Since this model evolved initially for single-core MCS, it suffers from two crucial weaknesses when applied to multi-core MCS. 1) As observed by [20], approaches adopting this model do not incorporate inter-task interference arising from accessing resources that are shared amongst cores such as memory buses, caches, and main memories in their scheduling or analysis. Experiments show that memory interference can contribute up to 300% to the WCET of a task [21], while the memory bus interference in commercial-off-the-shelf (COTS) systems can solely increase the WCET up to 44% [22]. As a consequence, we find it is of unavoidable necessity to account for this interference for multi-core MCS. 2) These approaches, upon switching to the degraded mode, do not provide any guaranteed service to lower-critical tasks. Since lower-critical tasks are still critical, industry criticizes this action as it may result in safety issues [13].

Fortunately, recent works address interference in MCS due to shared off-chip DRAMs [3, 23] and shared on-chip caches [24, 25]. Nonetheless, there is a limited focus on addressing the interference problem on shared buses in MCS. To our knowledge, [26] [20] are the only approaches to incorporate the memory bus interference in MCS modeling. However, both have certain limitations. [26] is adequate only for two criticality levels and mandates a particular mapping of tasks to cores. [20] comprises predictable COTS bus arbiters such as RR and FCFS, which lack the criticality notion. As a result, we find that the obtained bounds in [20] are pessimistic, foremost because of lacking a criticality-aware arbitration amongst different traffics on the memory bus.

In addition, we find that these limitations in [20,26] disallow them from exploring possible novel solutions at the arbiter level when it comes to the dynamic mode migrations of MCS.

2.1.1 Contributions

We address the interference problem on the shared memory bus in multi-core MCS by making the following contributions.

1. We expose strengths and inherent limitations of currently used arbiters in traditional single-criticality systems upon their applicability to MCS (Section 2.5).

2. Hence, we introduce CArb, an arbitration mechanism for controlling accesses to the shared memory bus in MCS (Section 2.6). CArb is a hierarchical two-tier arbiter that is, to our knowledge, the first to be criticality- and requirement-aware. This is necessary for two reasons. First, it results in optimal service allocation to tasks to meet their temporal requirements (Section 2.7). Second, it prioritizes tasks of higher criticality if the current set of memory requirements of all tasks is not schedulable. This is a vital characteristic when moving to higher modes in MCS (Section 2.8).

3. We illustrate a methodology to decompose WC memory access latencies from the WC computation latencies experienced by a task (Section 2.4). This has the advantage of allowing various MCS scheduling policies on cores to co-exist and operate in tandem with CArb; thereby, not imposing any restrictions on OS scheduling.

4. We propose two mechanisms to dynamically adapt the memory bus arbitration at run time to respond to increases in the monitored execution times of tasks (Section 2.8). We show how these mechanisms can mitigate these increases; thus, in some cases, postpone or even eliminate the system need to switch to a degraded mode. We believe that avoiding these switches is highly desirable because of their notoriously huge overheads. In addition, the proposed mechanisms prevent unnecessary suspension of lower-critical tasks.

5. We experiment with a case-study from the avionics domain as well as with synthetic experiments (Section 2.9). Our results show that CArb is well-suited for bus arbitration in multi-core MCS.

2.2 Related Work

There are several research efforts that investigate scheduling tasks with mixed criticalities on the same platform [14–18]. While earlier works primarily focus on single-core platforms [14, 15], recent efforts propose strategies for deploying MCS onto multi-core platforms [16–18]. An ongoing survey [27] maintains a comprehensive list of these efforts. There exist two practical issues in these efforts that are related to this thesis:

1. They suspend LO-critical tasks at the HI-mode [27], thus having no guarantees for the those tasks that are deemed low criticality, but still critical to some degree. As a result, industry is reluctant to widely adopt these efforts since the aforementioned suspension can result in safety issues [13].

2. They do not address temporal interference between tasks arising from accessing resources that are shared amongst cores such as memory buses, caches and DRAMs.

For the first issue, we are the first, to the best of our knowledge, to address it. In Chapter 2, we promote fine-grained rescheduling to enable higher-critical tasks to meet their new requirements, while not suspending the lower-critical ones, if possible (Section 2.8). *For the second issue*, as far as we are aware, [20, 26] are the only existing prior efforts to address it. The approach in [26] employs a software-based throttling mechanism to manage accesses to the shared main memory. It assigns a memory access budget to each core, and when a non-critical core exceeds its budget, [26] throttles it to guarantee requirements of the critical core. We find that this approach is suitable for only dual-criticality MCS, where each task is either critical or non-critical. For MCS with multiple criticalities, [26] faces the aforementioned issue of throttling lower-critical tasks. In addition, [26] mandates mapping all critical tasks to the same core. Two drawbacks arise from this requirement. 1) It limits the applicability of this technique to other scheduling approaches that do not meet this requirement. 2) Systems with large number of critical tasks cannot use this approach if critical tasks are not schedulable in a single core. The technique in [20] arbitrates amongst memory requests from all tasks using conventional round robin (RR) and first-come first-serve (FCFS) policies. However, these arbiters, as we discuss in Section 2.5, are agnostic to the distinct criticality and requirements of tasks, as they allocate the same service to all tasks. As a consequence, the bounds obtained in [20] are pessimistic. We address these limitations in Chapter 2 by proposing CArb that arbitrates accesses to the shared memory bus according to both criticality and timing requirements of all tasks. Utilizing CArb, we illustrate how to distinctly allocate service to tasks in an optimal fashion.

2.3 System Model

Figure 2.1 depicts the system considered in this chapter. We assume a multi-core system, where each core executes a single task. This task runs until completion on the dedicated core. We support any mapping of tasks to cores. This allows the integration of CArb with a wide variety of existing task mapping schemes. Existing cores share inter-core platform resources. Specifically, off-chip DRAM, on-chip last-level cache (LLC), and the memory bus connecting cores to the LLC. We assume that the interference on shared DRAM is resolved using existing techniques such as the partitioning scheme proposed in [23], or requirement-aware scheduling as we propose in [3] and detail in Chapter 4. Similarly, interference on shared data in the LLC is addresses by deploying cache partitioning or colouring [24]. Accordingly, this chapter focuses on the interference problem on the shared memory bus, and its impact on the total execution time of various tasks.

- We consider a mixed criticality system with n criticality levels. We classify tasks according to their criticality into groups that we denote as *classes*. Hence, there exist a set of n classes.
- Each class is defined as $C_l = \langle L_l, \Gamma_l \rangle$, where L is the criticality level and Γ_l is the total number of tasks in C_l. Higher values of L denote higher criticality levels.
- A task is characterized as: $\tau_{jl} = \langle L_l, T_{jl}, D_{jl}, E_{jl}, S_{jl}(L), I_{jl}(L), \Lambda_{jl} \rangle$ where:
  - T_{jl} is the minimum inter-arrival time of task jobs which represents the task period.
  - D_{jl} is the task deadline, where $D_{jl} = T_{jl}$.
  - S_{jl} is the WCET of any job of task τ_{jl} when τ_{jl} runs in isolation (no inter-task interference).

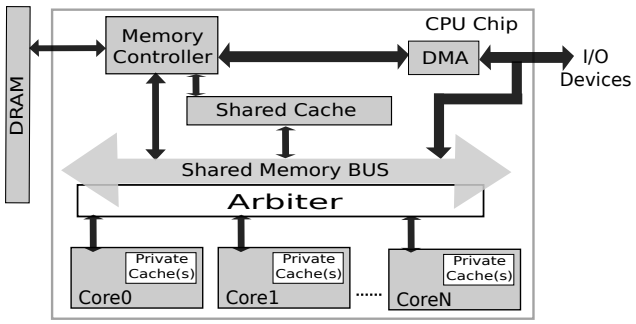


Figure 2.1: Multi-core architecture.

- Λ_{jl} is the maximum number of memory accesses issued by any job of τ_{jl}. It is worth noting that CArb makes no assumption about the memory access rate of tasks. Λ_{jl} represents the WC number of memory accesses per period over all periods of the task, similar to [28]. This is analogous to the execution time of the task. Both the number of memory accesses, and the execution are different from one period to another; nonetheless, the task model only considers the WC execution time of a task. Both S and Λ can be collected using either measurement-based techniques or static analysis tools. For sake of simplicity, we assume that Λ is constant for all CLs. In section 2.7.3, we generalize the model to consider $\Lambda(L)$ as function of the CL.
- I_{jl} is the WC additional latency due to inter-task interference. $I_{jl} = M_{jl} \times \Lambda_{jl}$, where M_{jl} is the WC interference delay per memory request. In Section 2.4, we present a methodology to calculate M_{jl}. Since the number of running tasks varies with regard to the CL, so does the interference amongst these tasks. Hence, I is a function in the CL, $I(L)$.
- E_{jl} is the total WCET. Since the interference delays can be considered to be additive to the task's WCET in isolation [29], we assume that $E_{jl} = S_{jl} + I_{jl}$. Each task has an S value for each CL in the system. This is a primary characteristic of MCS. The intuition behind these different values per task is as follows. The computed WC times of a task are estimates calculated using extensive testing and/or static analysis methods. Hence, based on the accuracy and pessimism levels of these methods, different estimates may exist. The higher the criticality level is, the more pessimistic the values are [9, 19].

This task model considers two extensions to the standard MCS model [27]. 1) We assume an arbitrary number of CLs. We promote this for two reasons. First, to not limit the integration support of CArb to only dual-criticality scheduling mechanisms; rather, CArb supports also mechanisms with more criticality levels such as [30]. Second, we encourage MCS models that adopt more criticality levels because current industrial standards, for instance in avionics domain, call for up to five levels. Examples of these standards include IEC 61508, DO-178B, DO-254 and ISO 26262 [19]. 2) Decomposition of total execution time, E, into S and I. This enables memory bus arbitration to optimize service allocation to tasks according to their deadline requirements. A more detailed discussion on this decomposition is in the following section.

2.4 Execution Time Decomposition

Migrating MCS onto multi-core platforms with inter-core shared resources, the interference delay due to shared resources amongst cores becomes an eminent component in the total WCET. Therefore, we claim that focusing on the interference delays of tasks is as necessary as the traditional focus on WCETs calculated in isolation. In consequence, we incorporate the total WCET E with its two components S and I in the proposed MCS model. This allows core scheduling techniques to focus on optimizations that affect S and shared memory arbitration techniques to minimize or eliminate I. In this chapter, we employ this decomposition process, and show that such separation enables attaining optimal solutions to the interference problem. This section illustrates the decomposition process, while Section 2.7.2 targets optimal allocations.

Existing approaches in scheduling MCS usually formulate the requirement on S relative to D as a schedulability condition. Tasks in the system are schedulable under the scheduling scheme only if they satisfy this condition. For the aforementioned reasons, we argue for substituting S with E in the schedulability analysis of multi-core MCS, and we assume S and I are additive such that $E = S + I$. Hence, if S is known beforehand, the schedulability condition turns into a requirement on the total interference delay encountered by each task such that the set of tasks is schedulable. Further, for static analysis purposes, the WC interference delay per memory request, M, must be assumed. Recall that the total number of memory requests of a task is Λ; thus, $I = M \times \Lambda$. Both S and Λ are predetermined task's characteristics. Subsequently, from the schedulability condition, we derive a requirement on M_{jl} for each τ_{jl} such that the set of tasks is schedulable. Since this chapter focuses on the interference on shared memory buses, we denote this condition as *the memory latency requirement*. The arbiter must allocate services to tasks such that the maximum memory latency of any request does not violate this requirement.

2.4.1 Illustrative Example

We show how to derive the memory access requirement from the schedulability condition. We use the partitioning algorithm proposed by Sha [31] as an example of a scheduling policy used in the avionics domain. The policy in [31] splits the set of tasks into partitions. All tasks of a partition must execute on the same core. A class, as defined in Section 2.3, can consist of multiple partitions. Hence, tasks of same class can execute in parallel on multiple cores if they belong to different partitions. In consequence, the partitioning algorithm of [31] resembles a general scheduling example by allowing tasks of same as well as different criticalities to run simultaneously; thus, interfere on shared memory. Under this algorithm, a time-division-multiplexing (TDM) scheduler assigns slots to partitions and a rate monotonic (RM) algorithm schedules tasks

of the same partition. Sha [31] proved that the sufficient schedulability condition for each partition is:

$$U_r \leq \gamma_r\big((\frac{2}{2-\hat{U}_r})^{\frac{1}{\gamma_r}} - 1\big). \tag{2.1}$$

Where for partition r, γ_r is the total number of tasks in r, U_r is the total utilization of those tasks, and $\hat{U}_r$ is the partial utilization granted to r, which is the number of TDM slots granted to the partition divided by the total number of slots in the TDM schedule. From this schedulability condition for a partition, we compute the memory latency requirements using the following procedure:

(1) Given that the utilization of all tasks in partition r is $U_r = \sum_{j=1}^{k_r} \frac{S_{jr}}{D_{jr}}$ and substituting S with $E = S + I$, as discussed earlier, then:

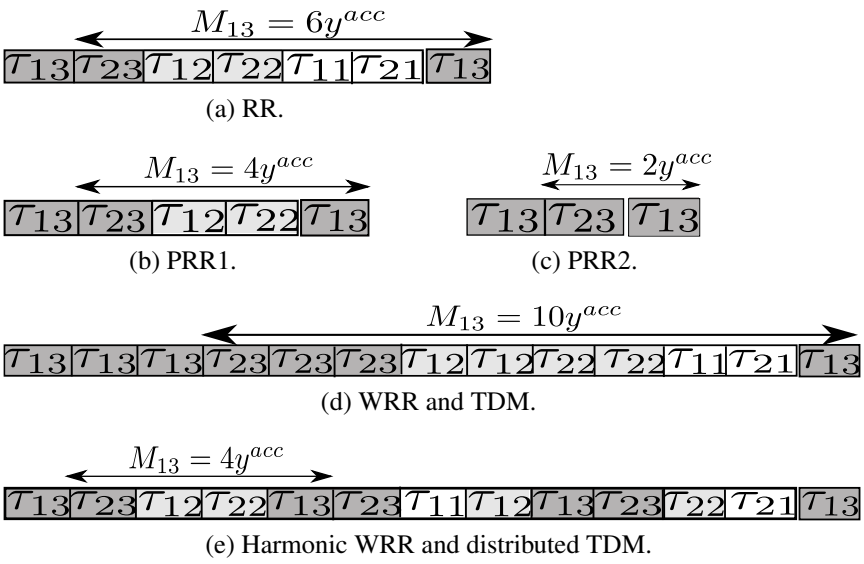
$$\sum_{j=1}^{\gamma_r} \frac{S_{jr} + M_{jr} \times \Lambda_{jr}}{D_{jr}} \leq \gamma_r\big((\frac{2}{2-\hat{U}_r})^{\frac{1}{\gamma_r}} - 1\big)$$

(2) Recall that S_{jr} and Λ_{jr} are predetermined for all tasks, then the memory access latency requirements per task to satisfy schedulability condition is obtained by Equation 2.2. Memory access latencies of all tasks of that partition must satisfy the condition in Equation 2.2. Notice that if a different scheduling algorithm is used, a similar procedure can be conducted to obtain the corresponding condition.

$$\sum_{j=1}^{\gamma_r} \frac{M_{jr} \times \Lambda_{jr}}{D_{jr}} \leq \gamma_r\big((\frac{2}{2-\hat{U}_r})^{\frac{1}{\gamma_r}} - 1\big) - \sum_{j=1}^{\gamma_r} \frac{S_{jr}}{D_{jr}} \tag{2.2}$$

2.5 Applicability of real-time arbiters in MCS

We study commonly used arbiters in traditional real-time systems to investigate their applicability on MCS. Particularly, we focus on RR arbiters: bare RR, prioritized RR (PRR), weighted RR (WRR), and harmonic RR (HRR) in addition to TDM arbiters: contiguous TDM, and work-conserving distributed TDM. We argue that an adequate arbiter for MCS must posse two features: *requirement-awareness* and *criticality-awareness*. Requirement-awareness implies that the arbiter is able to allocate service to tasks based on their temporal requirements. Comparatively, criticality-awareness is achievable when the arbiter allocates service to tasks relative to their criticality. We evaluate each arbiter with regard to adopting these two features using Figure 2.2 for illustration. In Figure 2.2, we assume a system with three criticality levels and 6 tasks.



(a) RR.

(b) PRR1.

(c) PRR2.

(d) WRR and TDM.

(e) Harmonic WRR and distributed TDM.

Figure 2.2: Real-time arbiters.

$\mathcal{T}_{13}$ and $\mathcal{T}_{23}$ are the highest critical, $\mathcal{T}_{12}$ and $\mathcal{T}_{22}$ are of medium criticality, and $\mathcal{T}_{11}$ and $\mathcal{T}_{21}$ are non-critical. y^{acc} is the access latency to the shared memory.

Bare RR. RR is dynamic and simple to implement. The arbiter equivalently rotates amongst tasks (Figure 2.2a). The WC latency of a request from any task is bounded by the number of tasks in the system; hence, RR assures predictability. RR allocates the same service to all tasks regardless of their distinct criticality and timing requirements. For instance, in Figure 2.2a, all tasks encounter the same WC latency of $6y^{acc}$ cycles. Hence, RR is neither criticality-aware nor requirement-aware; thus, ill-suited to MCS.

PRR. Authors in [32] address the deficiencies of RR by proposing PRR. The arbiter conducts RR arbitration amongst critical tasks only. Non-critical tasks gain access only on *slack slots*, which are slots when there are no ready requests from any critical task. This solution targets systems with dual-criticality. Applying PRR in MCS with more than two levels, critical tasks (but not the most critical) can be scheduled by two approaches. 1) They share the schedule with the most critical tasks; hence, attain as much service as them even though they may have different requirements. In Figure 2.2b, both tasks of C_2 and C_3 have a WC latency of $4y^{acc}$. 2) They share the slack slots with non-critical tasks; thus, they have no timing guarantees, and may miss their deadlines (Figure 2.2c). Accordingly, we find that PRR's applicability is limited to dual-criticality systems where tasks with the lower CLs have no requirements.

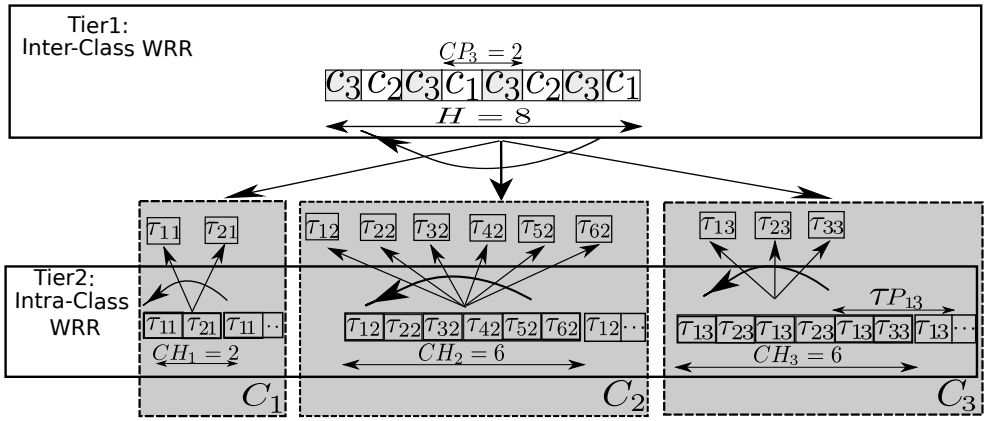
WRR and Contiguous TDM. Unlike bare RR, WRR [33] is capable of allocating different

amounts of service (slots or weights) to tasks based on their requirements as Figure 2.2d illustrates. Similar capability exists for contiguous TDM. The major difference between contiguous TDM and WRR is that TDM arbiters are, in general, non-work conserving. A slot assigned 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. On the other hand, WRR is work-conserving, and assigns idle slots to the first task with a ready request. Deploying either contiguous TDM or WRR, tasks with higher weights (or number of slots in TDM) encounter less average-case latency; though, the WC latency of requests from these tasks is either the same as or higher than bare RR. For example, in Figure 2.2d, the most critical task $\mathcal{T}_{13}$ obtains $1/4$ of the total slots. Notice that it suffers a WC latency of $10y^{acc}$ cycles compared to only $6y^{acc}$ cycles in bare RR. This is because in the WC, a request from any task (critical or non-critical) must wait for requests from all other tasks before it gets an access. Consequently, their deployment in MCS leads to pessimistic WCETs, and may not satisfy task requirements.

HRR and work-conserving distributed TDM. HRR [34] and work-conserving TDM [3] address this pessimism in WRR and contiguous TDM by evenly distributing slots assigned to tasks across the schedule as shown in Figure 2.2e. They have a different WC bound per task based on its requirements. Therefore, they are requirement-aware. Nevertheless, they assign service to tasks solely based on their timing requirements and not criticality. Upon applied to MCS, being non-criticality aware has two drawbacks. 1) In both approaches [3, 34], meeting the requirements for lower-criticality is as important as meeting those of the most-critical tasks. As aforementioned, in MCS, importance of fulfilling task requirements is relative to its criticality. For instance, in the automotive domain, it is crucial that the anti-lock brake system (ABS) meets its requirements over the proper functioning of the radio system does [35]. 2) Under the dynamic migration between various modes of the MCS system, a non criticality-aware approach is agnostic to which tasks must meet their requirements under all modes and which ones, on the other side, can be throttled at certain situations.

2.6 CArb: Proposed Arbitration Scheme

Motivated by the limitations of existing arbiters for MCS, we introduce CArb: a configurable criticality- and requirement-aware arbiter. CArb deploys a hierarchical two-tier arbitration scheme to manage accesses to the shared memory bus. It classifies tasks by their criticality grouping tasks of same CL in a class. Then, it executes a *harmonic WRR inter-class* arbitration among classes in the first tier, and a *harmonic WRR intra-class* arbitration amongst tasks of the same class in the second tier. Figure 2.3 depicts a system with 11 tasks classified into three classes C_1, C_2 and C_3, where C_3 is the most-critical. We use it as an example to illustrate CArb's operation.



(a) Inter- and intra-class arbitration.

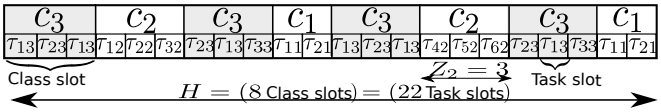
	CW	Z
C_3	4	3
C_2	2	3
C_1	2	2
H	8	

	τ_{13}	τ_{23}	τ_{33}	CH_3
TW	3	2	1	6

	τ_{12}	τ_{22}	τ_{32}	τ_{42}	τ_{52}	τ_{62}	CH_2
TW	1	1	1	1	1	1	6

	τ_{11}	τ_{21}	CH_1
TW	1	1	2

(b) Look-up table required for schedule parameters.



(c) Final CArb schedule.

Figure 2.3: Memory bus arbitration using CArb.

2.6.1 Inter-class Arbitration

CArb has two types of slots: *class slots* and *task slots*. A class slot consists of one or more task slots and is granted to a single class. The number of task slots in a class slot is generally distinct per class and is defined by its *window size*, Z_l. Since CArb deploys a harmonic WRR arbitration amongst classes, the number of class slots assigned to C_l is relative to its *class weight*, CW_l. *Schedule hyperperiod* is the summation of all class weights, $H = \sum_{l=1}^{n} CW_l$ such that CArb repeats the same schedule every H. Subject to CW_l and H, each class gets a slot every $CP_l = \frac{H}{CW_l}$, which we denote as CArb*'s class period*. In Figure 2.3, class weights CW_1, CW_2 and CW_3 are $2, 2$ and 4, respectively comprising a hyperperiod of $H = 8$, while the class window sizes Z_1, Z_2 and Z_3 are $2, 3$ and 3, respectively.

Algorithm 2.1 describes the inter-class arbitration mechanism. For each class slot, a flag bit is reset to indicate that the slot is not allocated yet (line 3). Then, the arbiter iterates through the set of classes starting from the most critical one. For each class, C_l, the arbiter checks if

C_l has to start a new period. If C_l starts a new period, a flag bit, denoted as *class grant* CG_l, is reset (line 6). If $CG_l = 0$, which implies that C_l is ready to be scheduled, and the current class slot is not allocated yet, CArb allocates this slot to C_l (lines 7 to 11). At this step, CArb moves to the intra-class arbitration to schedule tasks of C_l. Afterwards, CArb switches to the next class slot and repeats the same process again (the loop in lines 2 to 13) for H slots, then starts a new hyperperiod with same schedule. The inter-class WRR shown in Figure 2.3a exemplifies a schedule resulting from Algorithm 2.1, where the schedule repeats every 8 class slots.

Algorithm 2.1: CArb(. . .) – inter-class arbitration.

```
   Input: CW_l, Z_l, Γ_l ∀l in [1, n]
1  H ← SUM_∀l(CW_l);
2  foreach (classSlot in [0, H − 1]) do
3      allocated ← false;
4      foreach (l in [n, 1]) do
5          CP_l ← H/CW_l;
6          if (mod(classSlot, CP_l) = 0) then
7              CG_l ← 0;
8          end
9          if (CG_l = 0 and allocated = false) then
10             CG_l ← 1;
11             scheduleClass(Z_l, Γ_l);
12             allocated ← true;
13         end
14     end
15 end
```

2.6.2 Intra-class Arbitration

Recall that the inter-class tier grants Z_l task slots to C_l every class slot assigned to it. This results in a total of $CW_l \times Z_l$ task slots every H. The role of the intra-class arbitration is to distribute these task slots amongst tasks of C_l to satisfy their requirements. This is achieved by executing a per-class schedule that deploys a harmonic WRR arbitration amongst tasks of the same class. Thus, the *task weight*, $\mathcal{T}W_{jl}$, determines the number of task slots assigned to $\mathcal{T}_{jl}$. Summation of all task weights constructs the *class hyperperiod*: $CH_l = \sum_{j=1}^{K_l} \mathcal{T}W_{jl}$. The intra-class arbitration

Algorithm 2.2: scheduleClass(. . .) – intra-class arbitration.

```
   Input: Z_l, Γ_l
1      τW_jl ∀j in [1, Γ_l]
2  CH_l ← SUM_∀l τW_jl;
3  foreach (taskSlot in [0, Z_l − 1]) do
4      allocated ← false;
5      foreach (j in [1, Γ_l]) do
6          τP_jl ← CH_l/τW_jl;
7          if (mod(indx_l, τP_jl) = 0) then
8              τG_jl ← 0;
9          end
10         if (τG_jl = 0 and allocated = false) then
11             τG_jl ← 1;
12             inc(indx_l);
13             if (T_jl has a waiting request) then
14                 scheduleTask(T_jl);
15                 allocated ← true;
16             end
17         end
18     end
19 end
```

repeats the same task schedule amongst tasks of C_l every CH_l, while $\mathcal{T}_{jl}$ gets τW_{jl} task slots every CH_l. $\tau P_{jl} = \frac{CH_l}{\tau W_{jl}}$ is the CArb*'s task period* such that CArb must grant a task slot to $\mathcal{T}_{jl}$ every τP_{jl} task slots in CH_l. In Figure 2.3, weights of C_3's tasks τW_{13}, τW_{23} and τW_{33} are $3, 2$ and 1, respectively. This results in a class hyperperiod of $CH_3 = 6$.

Algorithm 2.2 illustrates the intra-class arbitration process. Clearly, it is very similar to the first tier arbitration amongst classes with some conceptual differences. The intra-class arbitration executes a distinct schedule per class— see for example the task schedules if C_1, C_2 and C_3 at the intra-class tier in Figure 2.3a. CArb tracks the number of task slots allocated to C_l in the current schedule hyperperiod by the counter $indx_l$ (line 7). For the current task slot, with particular $indx_l$ value, it checks if $\mathcal{T}_{jl}$ has to start a new period (line 4). It repeats this check for all tasks in C_l. $indx_l$ is reset at the start of every H. We dictate the task slot width to allow for one access to the shared memory finish. Hence, once CArb grants access to a request from any task, it cannot

be preempted. This is mandatory to guarantee predictability, while keeping the arbiter feasibly simple to implement. Having CArb implementing WRR, it is a work-conserving arbiter. For any task slot, if a task does have a ready request, CArb will allocate this slot to the first task in the schedule with a ready request (line 8 in Algorithm 2.2). The final schedule that CArb implements by executing both tiers of arbitration is akin to the instance shown in Figure 2.3c.

2.6.3 Area Overhead

For CArb to be able to execute a bus schedule satisfying memory requirements, it seeks the pre-knowledge of the variables that comprises this schedule, which we denote as *schedule parameters*. Particularly, it requires the values of $\mathcal{T}W_{jl}$, CW_l and Z_l for all classes and tasks. We formulate an optimization problem in Section 2.7 to specify the optimal values of these variables and solve this problem offline based on requirements obtained in Section 2.4. Hence, obtained schedule parameters are stored in a configurable look-up table during boot time. For the example schedule shown in Figure 2.3, we illustrate the look-up table structure in Figure 2.3b. Let each schedule parameter require a $32bits$ (or $4B$) register. Generally, for n classes and Γ_l tasks per class, class parameters (CW_l and Z_l) demand $(8 \times n)B$, while task parameters per class require $(4 \times \Gamma_l)B$. Accordingly, a total storage of $440B$ is sufficient to store the schedule of a system with 5 classes (5 is the maximum number of criticality levels specified by standards) and 100 tasks per class. We believe that this is a negligible area overhead for commodity multi-core systems.

2.7 WC Analysis and Problem Formulation

When using CArb, a request to the shared memory bus incurs two types of latencies, scheduling latency and access latency. Definitions 2.1 and 2.2 formally define these latencies.

Definition 2.1. ***Scheduling latency***, $y^{sch}_{jl,r}$, *of a request* req_r *generated by* $\mathcal{T}_{jl}$ *is measured from the time stamp of its issuance until it is granted access to the memory bus.* $y^{sch}_{jl,r}$ *is due to requests from other tasks scheduled before* $\mathcal{T}_{jl}$.

Definition 2.2. ***Access latency*** *is the latency suffered by a request generated by* $\mathcal{T}_{jl}$ *while it is accessing the shared memory. We assume that accessing the shared memory takes a fixed latency,* y^{acc}. *This latency can be considered as the WC access latency of the shared memory.*

2.7.1 WC analysis

Lemma 2.1. *The total WC latency of a memory request generated by τ_{jl}, denoted as y_{jl}^{tot}, is computed as follows.*

$$y_{jl}^{tot} = \left(\left(\sum_{v=1, v\neq l}^{v=\Gamma_l} \left\lceil \frac{\tau W_{vl}}{\tau W_{jl}} \right\rceil \right) + \left(\left\lceil \frac{\tau P_{jl}}{Z_l} \right\rceil \times \sum_{\substack{\forall e | (e \neq l \wedge \\ f_e \in \chi_l)}} \left(\left\lceil \frac{CW_e}{CW_l} \right\rceil \times Z_e \right) \right) + 1 \right) \times y^{acc}$$

Proof. The WC scheduling latency occurs when a request waits for the WC number of requests before it can get access to the resource. Recall that scheduling latency suffered by a request from τ_{jl} is due to requests from other tasks scheduled before τ_{jl}. These tasks belong either to the same class and cause *intra-class scheduling latency*, y_{jl}^{intraC}, or other classes and cause *inter-class scheduling latency*, y_{jl}^{interC}.

WC intra-class scheduling latency. In WC, during τP_{jl}, there are $\left\lceil \frac{\tau W_{vl}}{\tau W_{jl}} \right\rceil$ slots assigned to τ_{vl} ($v \neq l$). Since each task slot is y^{acc} and the number of tasks in C_l is Γ_l, the WC intra-class scheduling latency can be calculated as:

$$y_{jl}^{intraC} = \left(\sum_{v=1}^{v=\Gamma_l} \left\lceil \frac{\tau W_{vl}}{\tau W_{jl}} \right\rceil \right) \times y^{acc} \tag{2.3}$$

In Figure 2.3c, a request from τ_{33} has to wait for 3 requests from τ_{13} and 3 requests from τ_{23}.

WC inter-class scheduling latency. In WC, C_l has to wait for $DC_l = \text{MIN}(CP_l, n)$ distinct classes before it is granted a class slot, where each of these classes is assigned $\left\lceil \frac{CW_e}{CW_l} \right\rceil$ class slots. Furthermore, these classes are assigned the maximum number of task slots. Let $f_e = CW_e \times Z_e$ be the maximum number of task slots. In Figure 2.3c, $f_1 = 4$, $f_2 = 6$ and $f_3 = 12$. Equation 2.5 calculates the WC number of task slots CArb grants to other classed before it grants C_l a class slot. $\chi_l = \text{MAX}(F, DC_l)$ represents the largest DC_l elements of F where $F = \{f_1, f_2, ..., f_n\}$. χ_l identifies the DC_l classes with maximum number of task slots to represent the worst-case for C_l.

$$\sum_{\substack{\forall e | e \neq l \wedge \\ f_e \in \chi_l}} \left(\left\lceil \frac{CW_e}{CW_l} \right\rceil \times Z_e \right) \tag{2.4}$$

In addition, having C_l attained a class slot, does not necessarily imply that $\mathcal{T}_{jl}$ attains a task slot. Recall that once C_l attains a class slot, Z_l task slots are granted to its tasks, and $\mathcal{T}_{jl}$ gets a task slot every $\mathcal{T}P_{jl}$ at the C_l's schedule. Hence it gets one task slot every $\left\lceil \frac{\mathcal{T}P_{jl}}{Z_l} \right\rceil$ class slots granted to C_l. Consequently, a request from $\mathcal{T}_{jl}$ suffers from a WC inter-class scheduling latency of:

$$\left\lceil \frac{\mathcal{T}P_{jl}}{Z_l} \right\rceil \times \left(\sum_{\substack{\forall e | e \neq l \wedge \\ f_e \in \chi_l}} \left(\left\lceil \frac{CW_e}{CW_l} \right\rceil \times Z_e \right) \right) \times y^{acc} \tag{2.5}$$

In Figure 2.3c, a request from $\mathcal{T}_{33}$ has to wait in WC for $\left(\left\lceil \frac{CW_2}{CW_3} \right\rceil \times Z_2 \right) \times \left\lceil \frac{\mathcal{T}P_{33}}{Z_3} \right\rceil = (1 \times 3) \times 2 = 6$ task slot granted to other classes before it is granted an access.

Using Equations 2.3 and 2.5, and adding additional y^{acc} cycles to account for the access latency of the request itself, the total WC latency of any request is equal to the value computed in Lemma 2.1. □

2.7.2 Optimization problem formulation

Target Function. We formulate the schedule construction process as an optimization problem. The target is to generate the harmonic schedule with minimum hyperperiod that satisfies requirements of all tasks. Hence, the schedule is optimal amongst the set of harmonic schedules. Note that there may exist a non-harmonic schedule with a shorter schedule hyperperiod, which can be obtained using either unconstrained search or heuristic solutions (see for example [36]) than CArb. Since CArb is a hardware arbiter, we consider the harmonic property to minimize the area overhead as discussed in Section 2.6, while allowing for 100% bus utilization. We determine the optimal values of weights and window sizes assigned to classes and weights assigned to tasks to construct that schedule. Therefore, we express the target function as:

$$\text{MIN}(\sum_{l=1}^{l=n} CW_l \times Z_l).$$

Variables. The outcomes of this optimization problem are the task weights $\mathcal{T}W_{jl,}$ class weights CW_l, and class window sizes Z_l that construct the schedule. Accordingly, using these values, task periods $\mathcal{T}P_{jl,}$ class periods CP_l, class hyperperiods CH_l, and schedule hyper periods H are calculated.

Constraints. 1) The total WC latency must satisfy the memory access requirement obtained by Equation 2.2:

$$y_{jl}^{tot} \leq M_{jl} \tag{C.1}$$

2) Constraint C.2 prevents starvation at the inter-class arbitration tier. The lower bound, $CP_l \geq 2$, prohibits each class from starving other classes. If $CP_l = 1$, C_l will saturate the memory bus. The upper bound $CP_l \leq H$ prevents starving C_l as it assures that C_l will get at least one class slot in the schedule hyperperiod.

$$H \geq CP_l \geq 2 \tag{C.2}$$

3) Similarly, Constraint C.3 prohibits starvation at the intra-class arbitration tier:

$$CH_l \geq \tau W_{jl} \geq 2 \tag{C.3}$$

4) Three conditions are required to assert the periodicity characteristic such that CArb executes the schedule every H class slots or, equivalently every $CW_l \times Z_l$ task slots. First, the schedule hyperperiod, H, must be an integer multiple of CArb's class weights:

$$\frac{H}{CW_l} \in \mathbb{Z}_{>0}. \tag{C.4}$$

Second, every class hyperperiod, CH_l must be an integer multiple of CArb's task weights:

$$\frac{CH_l}{\tau W_{jl}} \in \mathbb{Z}_{>0} \quad \forall l \in [1, n] \forall j \in [1, \Gamma_l]. \tag{C.5}$$

Third, the total number of task slots granted to a class every H must be an integer multiple of the total number of required slots by tasks in that class:

$$\frac{CW_l \times Z_l}{CH_l} \in \mathbb{Z}_{>0}. \tag{C.6}$$

A final remark here is regarding constraint C.1. Recall that the condition on M_{jl} in Equation 2.2 depends on the value of S_{jl}, which is distinct per system mode. Two approaches can be followed based on two cases of the system requirements. 1) The first case is when the values of S for all tasks increase by the same ratio when system moves to higher levels. In this case, CArb stores only one optimal schedule that considers the value of S_{jl} for the lowest mode. Upon switching to higher modes, the operating system suspends lower-criticality tasks. Hence, their interference effect over all other tasks is eliminated. Since S increases by the same ratio for all tasks running at the new level, their schedule weights remain the same. Hence, the resulting schedule is sufficient to meet the requirements of the running tasks. 2) The second case is when the increase

ratios of S are not the same among tasks. In this case, CArb requires a schedule per each mode l that corresponds to $S_{jl}(l)$. Since current standards defines up to a maximum of 5 levels, the total area overhead of these schedules is approximately $2KBs$, which we find acceptable for commodity systems.

2.7.3 $\Lambda(L)$: The WC number of memory accesses as a function of CL

So far, we have considered Λ_{jl} to be fixed for all CLs. However, since Λ and S are calculated using same methods, either analysis or measurements, the level of assurance on Λ can, akin to S, depend on CL. Hence, the higher the criticality level, the larger the value of Λ for the same task. To address this situation, for each l-mode, we run the optimization framework considering $S_{jl'}(l)$ and $\Lambda_{jl'}(l)$ for all tasks at that level. Namely, tasks of $l' \geq l$ since tasks of $l'' < l$ are already suspended by the system at l-mode. A resulting schedule per mode needs to be stored at the boot time. Upon mode switching, CArb switches to the corresponding optimal schedule to fulfill the new requirements of all tasks executing at that mode.

2.8 Dynamic Re-arbitration

2.8.1 Motivation

Suppose that the system operates at l-mode. Let the execution time of τ_{jl+1} with criticality $l+1$, s_{jl+1}, exceed its WCET value, $s_{jl+1} > S_{jl+1}(l)$. Then, the conventional approach is to switch the system to $(l+1)$-mode suspending all tasks of l criticality. Tasks of criticality $l'' < l$ are already suspended at l-mode. This approach creates two challenges that motivate our proposed *fine-grained rescheduling at the arbiter level*. 1) Suspending l-critical tasks at the $(l+1)$-mode entails having no guarantees for those tasks. 2) Due to high overheads upon mode switching at the system scheduling mechanism as studied by [37], minimizing those switches is highly desirable.

2.8.2 Proposed Solutions

Leveraging CArb, we can conduct a set of fine-grained rescheduling techniques at the arbiter hardware that can mitigate the aforestated two issues of the conventional approach. We illustrate two of these techniques.

Scheme1: Prioritized CArb**.** This technique does not directly suspend tasks of l criticality. Instead, CArb allows them to access the shared memory bus only on slack slots when there are no ready requests from any task of higher criticality. As a consequence, this technique eliminates the interference from l-critical tasks. Thus, the interference suffered by τ_{jl+1}, i_{jl+1} decreases. Since the total execution time is $e_{jl+1} = s_{jl+1} + i_{jl+1}$, if the decrease in i_{jl+1} mitigates the observed increase in s_{jl+1} such that $e_{jl+1} \leq E_{jl+1}(l)$, there is no need to switch the mode. Otherwise, a mode switch is unavoidable. Since CArb schedule is statically predetermined, the maximum increase in the execution time of τ_{jl+1} that this technique can mitigate before switching the mode, denoted as s^{max}_{jl+1}, is known offline for all tasks. During running time, the operating system monitors the execution time of all tasks and makes the decisions shown in Algorithm 2.3.

Algorithm 2.3: Prioritized CArb.

if $(s_{jl+1} \leq S_{jl+1}(l))$ **then**
  run in normal l-mode;
end
else if $(S_{jl+1}(l) < s_{jl+1} \leq s^{max}_{jl+1}(l))$ **then**
  apply prioritized CArb;
end
else if $(s_{jl+1} > s^{max}_{jl+1}(l))$ **then**
  switch to $(l+1)$-mode;
end

Although prioritized CArb may appear similar to other priority-based arbiters, there are important differences. For instance, static-priority arbiters such as the one deployed in [23], does not provide any guarantees except for the highest-criticality tasks. In worst-case, tasks with the highest-criticality can issue requests forever; thus, starving other lower-criticality tasks. In other words, simple static-priority arbiters allow all tasks other than the highest-critical ones to issue requests only on slack time. In contrast, CArb applies prioritization only when a potential mode switch is discovered that CArb can avoid using re-arbitration. Amongst running tasks, only tasks of lowest-criticality (l-critical tasks at l-mode) issue requests on slack time. For example, assume a MCS with 5 criticality levels, where tasks are running at 1-mode, and the system monitors an increase in the execution time that CArb can mitigate without a mode switch. Accordingly, prioritized CArb forces tasks with 1-criticality to issue requests only on slack slots, and reallocate their slots to other tasks. All tasks of criticalities 2 to 5 are guaranteed to meet their requirements. On the other hand, if the system implements the aforementioned static-priority arbitration, only tasks of 5-criticality meet their requirements.

There exist other static-priority arbiters that avoid starvation of lower-criticality tasks by deploying budgeting mechanisms such as the credit-control static priority arbiter (CCSP) [38]. However, they have shortcomings when applied to MCS. For example, during each period, lower-criticality tasks have to wait for all higher-criticality tasks to finish their budgets before it can issue a single request. Accordingly, they may or may not meet their temporal requirements. This has the same disadvantage as the contiguous TDM discussed in Section 2.5. Contrarily, prioritized CArb distributes slots amongst running tasks in a harmonic way that is requirement-aware. This is true for all running tasks except for the lowest-criticality that CArb executes on slack time. In addition, once the monitored execution times decrease below their corresponding worst-case estimates, CArb reloads the normal schedule in the next schedule hyperperiod. Consequently, all tasks are guaranteed to meet their requirements.

Scheme2: Having an optimal schedule per mode. Prioritized CArb can be considered as a special case schedule, where weights of lower criticality tasks are set to 0. Although it successfully delays the mode switching, it can be considered a conservative solution. Generally, based on the amount of increase in the execution times, there exist a set of possible CArb solutions that can offset this increase. On l-mode, each solution comprises an optimal schedule that satisfy the new requirements, with larger $S_{jl'}$ values for all $l' > l$, while it maximizes the allocated slots to tasks of l instead of setting their allocated slots to 0. For the task under analysis, τ_{jl+1}, since the set of real execution times between S_{jl+1} and s^{max}_{jl+1} is uncountably infinite, some execution time values must be selected to find the optimal schedule for. In addition, the larger the selected execution times are, the less the allocated service to l-critical tasks. Hence, a trade-off exists between the allocated service to tasks of l, and the area required to store the selected number of corresponding schedules. For instance, suppose that only one additional schedule is stored for each level that corresponds to the middle point of value $s^{sch2}_{jl+1} = \frac{s^{max}_{jl+1}+S_{jl+1}}{2}$; hence, for l-mode and task τ_{jl+1}, the operating system decisions become as shown in Algorithm 2.4.

To obtain these additional optimal schedules for l-mode, Constraints C.1, and C.4 to C.6 apply only for $l' > l$ and the target function of the optimization problem changes to $\text{MAX}(CW_l \times Z_l)$ only for tasks of l criticality.
Finally, there are important observations to highlight. First, suppose that CArb is executing scheme1 or scheme2, if all monitored execution times decrease below their WCET values, CArb can move back to the original optimal schedule. Second, the overheads of proposed schemes are negligible compared to mode switches as 1) they are conducted at the arbiter hardware which is much faster than rescheduling at the system level, and 2) the operating system does not require to handle any of the complex procedures of mode switching; instead, it just sends a signal to CArb to move to one of these schemes. At the end of each hyperperiod, CArb monitors whether it receives this signal from the system. In case of signal reception, CArb applies the appropriate re-arbitration at the next schedule hyperperiod.

Algorithm 2.4: Scheme2 for dynamic re-arbitration.

if *($s_{jl+1} \leq S_{jl+1}(l)$)* **then**
  run in normal l-mode;
end
else if *($S_{jl+1}(l) < s_{jl+1} \leq s_{jl+1}^{sch2}$)* **then**
  apply optimal schedule2;
end
else if *($s_{jl+1}^{sch2} < s_{jl+1} \leq s_{jl+1}^{max}(l)$)* **then**
  apply prioritized CArb;
end
else if *($s_{jl+1} > s_{jl+1}^{max}(l)$)* **then**
  switch to $(l+1)$-mode;
end

2.8.3 Effect of Re-arbitration on Lower-criticality Tasks

Under normal operation, a MCS should satisfy requirements of both higher and lower criticality tasks. However, when the execution time of tasks increase, this may not be possible. The objective of the dynamic re-arbitration is to achieve the following two goals at each criticality level:

1. Guarantee the timing requirements of higher criticality tasks.
2. Provide lower criticality tasks with the maximum possible service after satisfying the first goal.

These two goals do not guarantee satisfying the timing requirements of lower-criticality tasks upon re-arbitration. In fact, upon re-arbitration, CArb will reduce the service delivered to lower-criticality tasks to satisfy requirements of higher ones. Nonetheless, the proposed fine-grained re-arbitration, unlike the traditional mode-switching approach, does not completely suspend lower-critical tasks unless needed. This is important since lower-criticality tasks are usually soft-real time tasks, which care about average-case rate of service or memory bandwidth. Hence, degrading their service is potentially a more practical solution than completely suspending them [13].

Use-case requirements					OS Scheduling using [31]			Optimal CArb parameters	
τ_{jl}	$D_{jl}\ (ms)$	$S_{jl}\ (ms)$	Λ_{jl}	Partition	Core	U	memory access requirements	τw_{jl}	(CW_l, Z_l)
τ_{14}	25	1.06	500	1	1	0.25	$M_{14} \leq 5.02\mu s$	6	$(3,4)$
τ_{24}	50	3.09	500	2	1	0.25	$M_{24} \leq 8.11\mu s$	3	
τ_{34}	100	2.7	500	3	1	0.25	$M_{34} \leq 23.17\mu s$	1	
τ_{44}	200	1.09	500	4	1	0.25	$M_{44} \leq 45.96\mu s$	2	
τ_{13}	25	0.94	1000	5	2	0.4	$2M_{13} + M_{23} + M_{33} \leq 6.45\mu s$	6	$(6,4)$
τ_{23}	50	1.57	1000					4	
τ_{33}	50	1.68	1000					4	
τ_{43}	50	4.5	1000	6	2	3/5	$4M_{43} + 4M_{53} + 2M_{63} + M_{73} \leq 35.28\mu s$	3	
τ_{53}	50	2.94	1000					3	
τ_{63}	100	1.41	1000					3	
τ_{73}	200	6.75	1000					1	
τ_{12}	50	5.4	4000	7	3	0.4	$M_{12} \leq 1.77\mu s$	1	$(3,1)$
τ_{11}	50	2.4	2000	8	3	0.6	$4M_{11} + M_{21} + 4M_{31} + M_{41} \leq 28.77$	5	$(12,5)$
τ_{21}	200	0.94	2000					2	
τ_{31}	50	1.06	2000					5	
τ_{41}	200	2.28	2000					3	
τ_{51}	25	4.75	3000	9	4	1	$8M_{51} + 2M_{61} + M_{71} + 2M_{81} + 4M_{91} \leq 24.17$	20	
τ_{61}	100	12.87	3000					6	
τ_{71}	200	0.47	3000					3	
τ_{81}	100	1.24	3000					6	
τ_{91}	50	1.62	3000					10	

Table 2.1: Experiment using the avionics use-case from Honeywell [9].

2.9 Experimental Evaluation

We experimentally prototype CArb using a multi-core architectural simulator called MacSim [39] with CArb managing accesses to a shared L3 cache. We use a multi-core architecture of four x86 cores, private 16KB L1 and 256KB L2 caches per core, and a single 1MB L3 cache shared and partitioned amongst cores and operates at 1GHz. The access latency of the L3 cache is 50 cycles. The evaluation consists of four parts as follows.

1. We evaluate CArb using a real use-case MCS requirements from the avionics domain.

2. We compare CArb with commonly used real-time arbiters (namely, RR and different configurations of WRR).

3. We highlight the trade-offs associated with adapting CArb's schedule parameters.

4. We study the effectiveness of the proposed re-arbitration schemes.

2.9.1 Avionics Use-case

Experiment setup. We simulate a workload of 21 tasks derived from partition-based avionics system from Honeywell [9]. Table 2.1 tabulates for each task: the deadline, D_{jl}, WC execution time in isolation, S_{jl}, and the maximum number of memory access issued in a period, Λ_{jl}. The workload has 9 partitions (column 5 in Table 2.1) and 4 criticality classes, C_1 to C_4. Since the actual task implementations are not publicly available, we implement in-house workloads that match requirements of these tasks. We deploy the algorithm proposed by Sha [31] to schedule tasks on cores using core assignments and utilizations given in columns 6 and 7 of Table 2.1, respectively. ***Obtaining*** CArb ***parameters.*** We use the schedulability condition in Equation 2.2 to construct the memory latency, which we show in column 8 of Table 2.1. Afterwards, we implement the optimization framework proposed in Section 2.7 in Matlab to obtain the optimal values of CArb's schedule parameters. According to the memory latency requirements, we get the optimal values for $\mathcal{T}W_{jl}$ (column 9 in Table 2.1), CW_l and Z_l (column 10) that satisfy these requirements while minimizing the schedule hyperperiod.

Results. Figure 2.4 shows the WC latencies obtained when CArb executes the optimal schedule to arbitrate requests from all tasks to the shared L3 cache. Clearly, the values satisfy all the memory access latency requirements in column 8.

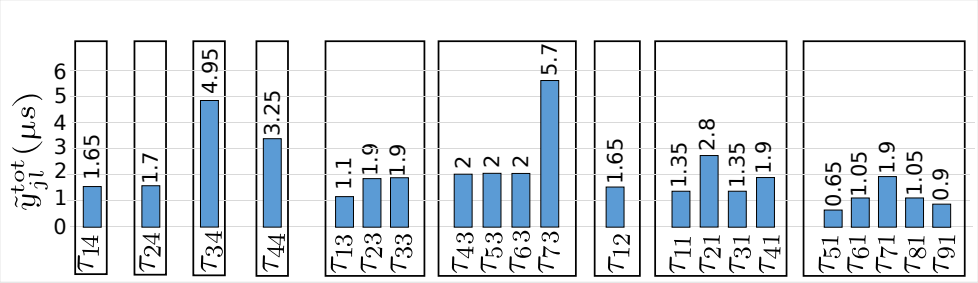


Figure 2.4: Avionics use-case results.

2.9.2 Minimum Achievable Latency

In this experiment, we show the usefulness of CArb's requirement-awareness in satisfying memory requirements that are unsatisfiable by conventional non-requirement-aware arbiters such as RR and WRR.

Experiment Setup. Towards this target, we vary the number of co-running critical and non-critical tasks and plot the minimum possible worst-case latency that can be guaranteed to a critical task. We call this latency the minimum achievable latency because the arbiter cannot guarantee a lesser latency for the critical task under any possible configuration. If a critical task has a memory requirement, M_{jl}, that is less than this minimum achievable latency, the set of tasks is deemed unschedulable by the arbiter. Figure 2.5 shows the results of this experiment. WRR-2 and WRR-4 represent a WRR arbitration, where each critical task is granted two and four consecutive slots, respectively. both arbiters grant one access slot for each non-critical task. The access latency of one request is 50 cycles.

Observations. 1) CArb is able to grant the critical task a fixed service regardless of the number co-running number of tasks. Varying the number of non-critical tasks from 2 to 8 has no effect on the minimum achievable WCL in Figure 2.5. For the case of 0 non-critical tasks, the minimum achievable WCL is less since there are no slots at all granted to (and hence, no interference from) the non-critical tasks. 2) The minimum achievable WCL increases for both RR and WRR by increasing the number of co-running tasks whether they are critical or not. This is because each task has to get at least one access slot. Since the slots are assigned contiguously, these slots contribute to the total WCL of the critical task. 3) In Figure 2.5, by increasing the number of co-running critical tasks, the minimum achievable WCL increases since we assume all critical tasks have the same requirements.

2.9.3 Synthetic Experiments

CArb is capable of meeting various system requirements by adapting its configurable parameters CW_l, Z_l and $\mathcal{T}W_{jl}$. Certainly, this adaptation involves a trade-off between requirements of different tasks. It is the role of the optimization framework to explore this trade-off and provide optimal values that satisfy all requirements. However, this experiment does not focus on discovering the optimal setting, but giving the reader a perspective on how parameters influence the outcome. In doing that, we disable the optimization framework and sweep each parameter to study its effect.

Experiment setup. We assume a system with three classes C_3, C_2 and C_1. C_3 has two tasks $\mathcal{T}_{13}$ and $\mathcal{T}_{23}$, C_2 has one task $\mathcal{T}_{12}$ while C_1 has four tasks $\mathcal{T}_{11}$ to $\mathcal{T}_{41}$. We run each task on a core

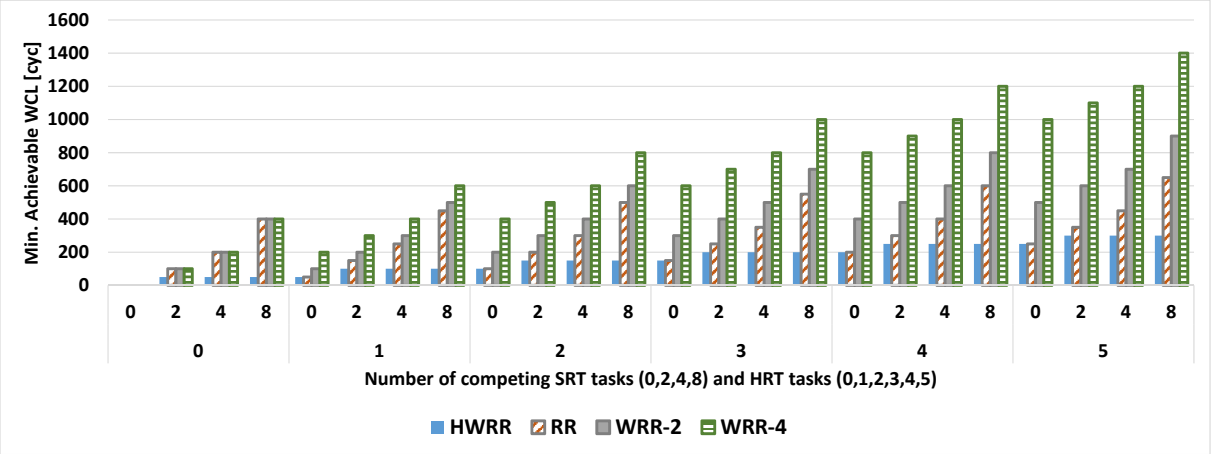


Figure 2.5: Latency requirements.

Exp.	CW_3	CW_2	CW_1	Z_3	Z_2	Z_1	τW_{j3}	τW_{j2}	τW_{j1} $j \neq 4$	τW_{41}
1	1	1	vary	2	1	4	1	1	1	1
2	2	2	4	1	1	vary	1	1	1	1
3	2	2	4	1	1	3	1	1	1	vary

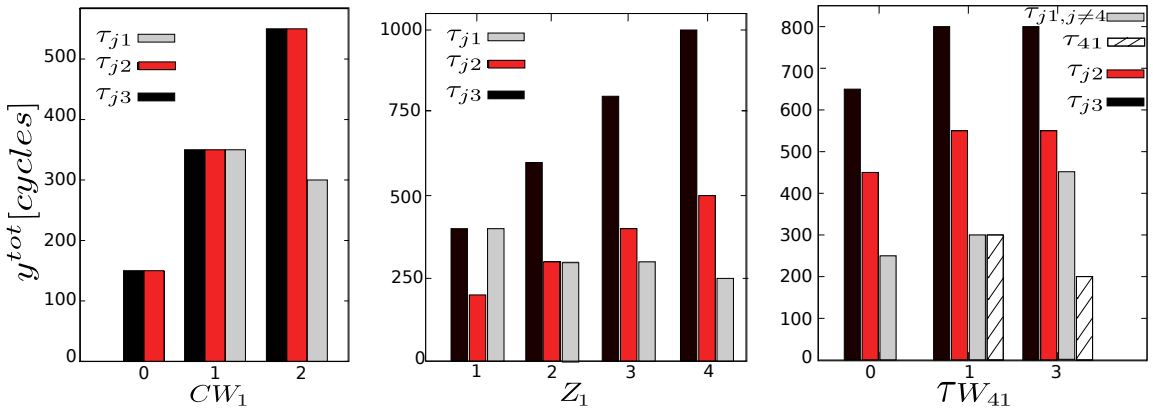
Table 2.2: Parameters of synthetic experiments.

and CArb manages accesses to a shared cache among cores. We conduct three experiments to: 1) vary CW_1, 2) vary Z_1, and 3) vary τW_{41}. Table 2.2 shows the values of all parameters used in these experiments. Figure 2.6 delineates the results of each experiment.

Observations. 1) Increasing CW_1, CArb grants more class slots to C_1. Similarly, increasing Z_1 will increase the number of task slots assigned to C_1's tasks. Consequently, in both cases, y^{tot} of C_1's tasks will decrease at the expense of increasing y^{tot} of tasks in C_2 and C_3. We also show the amount of interference that C_1's tasks contribute to latencies of tasks belonging to other classes by illustrating the case when no task from C_1 is scheduled ($CW_1 = 0$). Since this situation implies starving C_1's tasks, the optimization framework prohibits it under normal conditions.

2) At certain values, increasing weights or window sizes of a class may not decrease y^{tot} of

tasks in that class. For example, in Figure 2.6b, increasing Z_1 from 2 to 3 does not decrease y_{j1}^{tot}, while it has a negative effect on y^{tot} of tasks in C_2 and C_3. The rationale behind this observation is that increasing Z_1 from 2 to 3 do not, in fact, change the WC situation for tasks in C_1. According to the values of experiment 2 in Table 2.2 and $Z_1 = 2$, each task in C_1 attains 2 of 12 task slots in the schedule hyperperiod. Therefore, it has a WC scheduling latency of $\lceil \frac{12}{2} \rceil = 6$ slots. Increasing Z_1 from 2 to 3, each task in C_1 wins 3 of 16 task slots; hence, the WC scheduling latency becomes $\lceil \frac{16}{3} \rceil = 6$ slots. As a consequence, increasing Z_1 from 2 to 3 does not change y_{j1}^{tot}; however, it decreases average-case latency as tasks of C_1 execute more frequently.



(a) Class weight. (b) Class window size. (c) Task weight.

Figure 2.6: Synthetic experiments (y-axis is the total WCL, y^{tot}).

3) By changing $\mathcal{T}W_{41}$ (Figure 2.6c), the intra-class schedule of C_1 changes. Apparently, increasing $\mathcal{T}W_{41}$ decreases y_{41}^{tot} at the expense of increasing y^{tot} of other tasks in C_1. Notice that with the exception of $\mathcal{T}W_{41}= 0$, changing $\mathcal{T}W_{41}$ has no effect on y^{tot} of tasks in C_2 and C_3. This is because the inter-class schedule remains the same. This is a consequence of the criticality-awareness of CArb as it separates class arbitration from task arbitration. Since assigning $\mathcal{T}W_{41}= 0$ will result in a free slot that will be assigned to other tasks, tasks of C_2 and C_3 have less y^{tot}.

2.9.4 Dynamic Re-arbitration

Experiment Setup. In this experiment, we investigate the capabilities of CArb's dynamic re-arbitration mechanisms proposed in Section 2.8. We use the parameters in Table 2.3 to simulate a system with 3 classes. The partitioning algorithm in Sha [31] is used for core scheduling and

Table 2.3 tabulates the used partitions and utilizations. According to the standard MCS model,

τ	$D\ (ms)$	$S\ (ms)$	Partition	Core	U	Λ	$M(\mu s)$	τw	CW	Z
τ_{13}	5	1	1	1	0.5	2000	$M_{13} \leq 0.22$	1	4	1
τ_{23}	5	1	2	1	0.5	2000	$M_{23} \leq 0.22$	1		
τ_{12}	5	2	3	2	1	1000	$2M_{12} + M_{22} \leq 1.28$	1	2	1
τ_{22}	10	3				1000		1		
τ_{11}	10	2	4	3	1	2000	$3M_{11} + 2M_{21} \leq 2.9$	1	2	1
τ_{21}	15	8				2000		1		

Table 2.3: Parameters of the dynamic case experiment.

there are 3 modes of operations. 1-mode is the normal mode, where all tasks of all classes are operating according to the requirements given in Table 2.3. In 2-mode, the operating system suspends tasks of C_1 and only tasks of classes C_2 and C_3 are utilizing the hardware. Finally, in 3-mode, tasks of C_1 and C_2 are suspended and only tasks of C_3 have the permit to execute. Normally, a switch from the 1-mode to the 2-mode occurs when the execution time of any task in C_2 or C_3 exceeds its S_{ij} value in Table 2.3. To expose benefits of CArb dynamic re-arbitration, we postpone this mode switch and investigate if this re-arbitration is able to mitigate the increase in the execution time such that the requirements of tasks in C_2 and C_3 are met without suspending tasks of C_1. We model the increase in the execution time by decreasing the core operating frequency.

Observations. All tasks are affected by the frequency scaling. For clarity, we focus on results of C_2's tasks (partition 3). Figure 2.7 depicts $U_3 = \sum_{j=1}^{2} \frac{e_{j3}}{D_{j3}} = \sum_{j=1}^{2} \frac{s_{j3} + m_{j3} \times \Lambda_{j3}}{D_{j3}}$. The dotted line is the schedulability bound (right hand side of Equation 2.1).

1)Deploying CArb *without dynamic re-arbitration and disable mode switching*. As expected, decreasing the frequency, s_{12} and s_{22} increase and U_3 keeps increasing until violating schedulability condition (*noDynamic* plot in Figure 2.7).

2) Deploying CArb *with scheme1*. When s_{12} and s_{22} exceed their corresponding WCETs, S_{12} and S_{22}, CArb switches to the prioritized CArb mechanism, where tasks of C_1 gains access only on slack slots. As Figure 2.7 illustrates, *Scheme1* mitigates up to 12% and 18% increase percentages in s_{12} and s_{22}, respectively, without requiring the operating system to switch to mode 2. This results in postponing the mode switch from the frequency point of 990MHz to 950MHz. However, this comes at the expense of switching all tasks of C_1 to execute on slack slots.

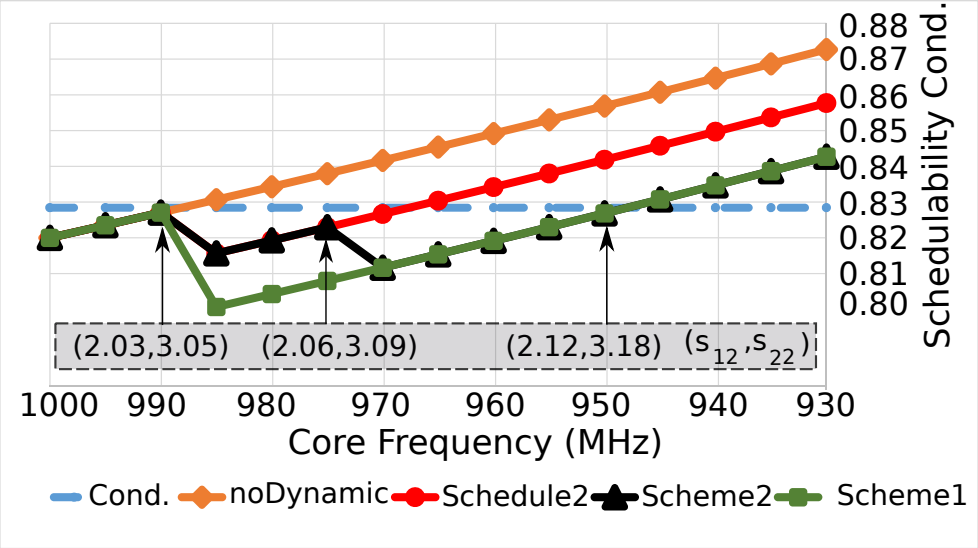


Figure 2.7: Effect of decreasing core frequency on tasks of C_2.

3) Deploying CArb *with scheme*2. Given the trade-off discussed in Section 2.8, we choose to store only one additional schedule configuration for scheme2 per mode. We choose a middle point between the WCETs and the maximum execution times that scheme1 can mitigate, where (s_{12}, s_{22}) equal $(2.06, 3.09)ms$, respectively, in Figure 2.7. This point is statically predetermined and we rerun the optimization framework to obtain the new optimal schedule. The obtained optimal schedule does not change the intra-class schedule and only reallocates the inter-class slots. This is achievable exclusively because CArb is criticality-aware with hierarchical scheduling. In the new schedule (*Schedule*2 in Figure 2.7), the class weights are the same as in Table 2.3, while class window sizes change to $1, 2$ and 2 for C_1, C_2 and C_3 respectively. Instead of directly applying prioritized CArb once the execution times exceeds their WCETs, *Schedule*2 mitigates increases up to 6% and 9% in s_{12} and s_{22}, respectively. This occurs from 990MHz to 970MHz in Figure 2.7. In addition, it guarantees some service allocation on the memory bus to C_1's tasks. Afterwards, *Scheme*2 deploys the prioritized CArb (from 970MHz to 950MHz in Figure 2.7). Finally, a mode-switch is unavoidable when s_{12} and s_{22} exceed the point $(2.12, 3.18)ms$ (region after 950MHz in Figure 2.7).

2.10 Summary

In this chapter, we addressed the inter-task interference problem in multi-core mixed criticality systems by presenting CArb. CArb is a criticality- and requirement-aware bus arbiter adopting two-tier weighted round-robin arbitration. CArb has the following advantages: it does not restrict the scheduling policy for tasks on cores, and it supports any number of criticality levels. In addition, it optimally allocates service to tasks through configurable schedules loaded at boot time. CArb is capable to dynamically adapt its schedule under varying system conditions. This adaptation proves its effectiveness to mitigate the system need to switch to a degraded mode upon increases in the execution times of tasks. Finally, we evaluated CArb using avionics case-study and synthetic experiments.

Chapter 3

PMSI: Predictable Cache Coherence for Multi-core Real-time Systems

This chapter addresses the challenge of allowing simultaneous and predictable accesses to shared data on multi-core systems. We accomplish this by proposing a predictable cache coherence protocol, which mandates the use of certain invariants to ensure predictability. In particular, we enforce these invariants by augmenting the classic MSI protocol with transient coherence states, and minimal architectural changes. This allows us to derive worst-case latency bounds on predictable MSI (PMSI) protocol. We implement PMSI in gem5, and execute SPLASH-2 and synthetic multi-threaded workloads. Our empirical results show that our approach is always within the analytical worst-case latency bounds, and that PMSI improves average-case performance by up to $4\times$ over the next best predictable alternative. PMSI has average slowdowns of $\mathbf{1.45}\times$ and $\mathbf{1.46}\times$ compared to conventional MSI and MESI protocols, respectively.

3.1 Introduction

In HRT systems, correctness depends not only on the functioning behavior, but also on the timing of that behavior [40]. Applications running on these systems have strict requirements on meeting their execution time deadlines. Missing a deadline in a HRT system may cause catastrophic failures [32]. Therefore, ensuring that deadlines are always met via static timing analysis is mandatory for such systems. Timing analysis computes an upper bound for the execution time of each running application on the system by carefully accounting for hardware implementation details, and using sophisticated abstraction techniques. The WCET of that application has to be less than or equal to this upper bound to achieve predictability. As application demands continue

to increase from the avionics [41] and automotive [42] domains, there is a surge in wanting to use multi-core platforms for their deployments. This is primarily due to the benefits multi-core platforms provide in cost, and performance. However, multi-core platforms pose new challenges towards guaranteeing temporal requirements of running applications.

One such challenge is in maintaining coherence of shared data stored in private cache hierarchies of multi-cores known as *cache coherence*. Cache coherence is realized by implementing a protocol that specifies a core's activity (read or write) on cached shared data based on the activity of other cores on the same shared data. While cache coherence protocols can be implemented in software or hardware, modern multi-core platforms implement the cache coherence protocol in hardware. This is so that software programmers do not have to explicitly manage coherence of shared data in the application. A recent work studied the effect of cache coherence on execution time using five different Intel and AMD processors and three coherence protocols [43]. The study compared between executing an application sequentially and in parallel. It concluded that the interference from cache coherence can severely reduce benefits gained from parallelism. In fact, it can make the parallel execution 3.87$\times$ slower than sequential execution.

This emphasizes the importance of considering cache coherence effects when deriving WCET bounds. However, as observed by a recent survey [44], there is no existing technique to account for the effects of coherence protocols in static timing analysis in real-time systems. As a result, tasks on multi-core systems cannot coherently and predictably share data unless some restrictions to eliminate those effects are enforced. For instance, a possible solution is disabling the caching of shared data [45, 46]. Clearly, this solution significantly increases the execution time of applications with shared data, which may render applications unschedulable. Another solution prohibits tasks with shared data from running simultaneously on different cores using task scheduling [47]. However, this solution requires special hardware performance counters and modifications to currently available scheduling techniques. A third solution suggests modifying the applications by marking instructions with shared data as critical sections such that they are accessed by only a single core at any time instance [48]. Although this allows caching of shared data, it stalls all tasks but one from accessing the data, which in WC amounts to sequentially running the tasks.

In summary, existing solutions prohibit tasks from simultaneously accessing shared data. This approach successfully avoids data incoherence, but it does so at the expense of one or more of the following: 1) severely degrading performance, 2) imposing scheduling restrictions, 3) imposing source-code modifications, and 4) requiring hardware extensions. This chapter presents a predictable cache coherence protocol to allow for simultaneous accesses to shared data. The proposed solution provides considerable performance improvements, does not impose any scheduling restrictions, and does not require any source-code modifications.

3.1.1 Contributions

We address the problem of maintaining cache coherence in multi-core real-time systems by modifying current coherence protocols such that data sharing is viable for real-time systems in a manner amenable for timing analysis. Doing so, we make the following contributions:

1. We identify behaviors in conventional coherence protocols that can lead to unpredictability. The identified behaviors are general and independent of the implementation details of the deployed cache coherence protocol. We use these observations to propose a set of invariants to address unpredictability in coherence protocols.

2. We show the unpredictable behaviors and the proposed solutions to rectify them using the MSI coherence protocol. Accordingly, we propose predictable MSI (PMSI), a coherence protocol that fulfills the proposed invariants on MSI by introducing a set of transient states while making minimal architectural changes to cache controllers (Section 3.6). We release the implementation of PMSI at [49] for researchers to use and extend. Modern commodity multi-core architectures implement various protocols that are optimizations of MSI such as MESI, MOESI and MESIF [50, 51]. Accordingly, our observations and proposed invariants are applicable to modern cache coherence protocols with those optimizations.

3. We provide a timing analysis for our proposed coherence protocol and decompose the analysis to highlight the contributions to latency due to arbitration logic and communication of coherence messages between cores (Section 3.7).

4. We evaluate the proposed coherence protocol using the gem5 full-system simulator [52] (Section 3.8). Our evaluation shows that cache coherence can increase the memory latency up to $10\times$ in a quad-core system. This further emphasizes the importance of providing safe bounds that account for the effect of cache coherence. Performance evaluation shows that PMSI achieves up to $4\times$ speedup over competitive approaches.

3.2 Related Work

Sharing data coherently has been investigated for many years for conventional performance-oriented architectures. Researchers have proposed different coherence protocols that are historically classified into directory-based or snoopy-based [10]. Recently, other alternatives have been proposed such as the token-coherence [53] and time-based coherence [54]. Snoopy-based protocols requires a totally ordered interconnection network [10]. Some approaches have been proposed to deploy snoopy-based protocols into unordered networks [55] and split-transaction

buses [56]. The latter is the most related to this thesis for two reasons: 1) we assume a bus-based architecture, and 2) usually predictable arbiters force a different ordering than the arrival order to ensure predictability. Thus, we discuss the solution proposed in [56] and highlight why it is ill-suited for real-time systems. The approach followed by the POWERpath-2 bus architecture in the SGI multiprocessor chip [56] is to disallow multiple pending requests to the same cache line. This is achieved by maintaining a pending-read queue of size 8 to track read requests that are not serviced yet. Any request (write or read) that targets a cache line matching that targeted by one of the pending reads is stalled until the read is serviced. This solution is not suitable for real-time systems because it does not guarantee a bounded memory latency. For example, the number of serviced requests from one core is uncontrolled; therefore, one core can flood the pending-read queue resulting in stalling all other cores.

For real-time systems, recent research efforts investigated the access latency overhead resulting from shared buses [32], caches [24,57,58], and dynamic random access memories (DRAMs) [3, 59, 60]. Data isolation appears to be the prevalent technique in literature. For shared caches, most of these efforts primarily focused on preventing a task's data accesses from affecting another task's data accesses. They used data isolation between tasks by utilizing strict cache partitioning [24] or locking mechanisms [57]. Authors in [58] promote splitting the data cache into multiple data areas (e.g. stack, heap,..etc.) to simplify the analysis. However, they indicate that the coherence is still an issue that has to be addressed. Similarly, several proposals for shared main memories deployed data isolation by assigning a private memory bank per core [59, 60]. However, we find that data isolation suffers from three limitations. 1) They disallow sharing of data between tasks; thus, disabling any communication across applications or threads of parallel tasks running on different cores. 2) It may result in a poor memory or cache utilization. For instance, a task may keep evicting its cache lines if it reaches the maximum of its partition size, while other partitions may remain underutilized. 3) It does not scale with increasing number of cores. For example, the number of cores in the system has to be less than or equal to the number of DRAM banks to be able to achieve isolation at DRAM. Recent works [3, 61, 62] recognized these limitations, and offered solutions for sharing data. Authors in [3] share the whole memory space between tasks for main memory, and [61, 62] suggested a compromise by dividing the memory space into private and shared segments for caches. Nonetheless, these approaches focus on the impact of sharing memory on timing analysis, and they do not address the problem of data correctness resulting from sharing memory. Authors of [63] study the overhead effects of co-running applications on the timing behaviour in the avionics domain, where coherence is one of the overhead sources. A recent survey [44] observed that there is no existing technique to include the effects of data coherence on timing analysis for multi-core real-time systems. However, there exist approaches that attempt to eliminate unpredictable scenarios that arise from data sharing. Authors in [47] proposed data sharing-aware scheduling policies that avoid running tasks with

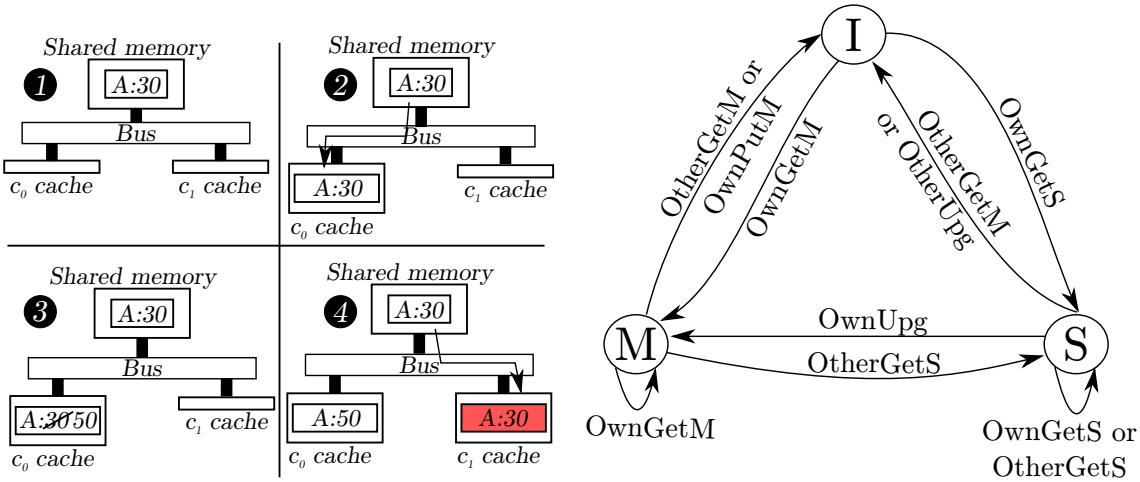
shared data simultaneously. A similar approach proposed by [43] redesigned the real-time operating system to include cache partitioning, task scheduling, and feedback from the performance counters to account for cache coherence in task scheduling decisions. Such approaches rely on hardware counters that feed the schedule with information about memory requests. They also require modifications to existing task scheduling techniques. For example, the solution in [47] is not adequate for partitioned scheduling mechanisms. A different solution introduced in [48] applied source-code modifications to mark instructions with shared data as critical sections. These critical sections were protected by locking mechanisms such that they were accessed only by a single core at any time instance. This solution suffers from certain limitations. 1) It exposes cache coherence to the software to assure correctness of shared data. 2) Only one core can accesses a cache line of shared data at a time. Other cores requesting this data have to stall. In worst case, this is equivalent to sequential execution. 3) Additionally, they still require hardware to keep track of whether each cache line is shared or not.

We address these limitations in Chapter 3 by proposing PMSI. PMSI allows tasks to simultaneously and predictably access shared data, which considerably improves performance. In addition, it does not pose any requirements on task scheduling techniques, and it does not require software modifications. We also provide a timing analysis that accounts for memory coherence for PMSI.

3.3 Background: Cache Coherence

The objective of cache coherence is to provide all cores read access to the most recent write on shared data. Incoherent sharing of data occurs when multiple cores read different versions of the same data that is present in their private cache hierarchies. Figure 3.1a shows one instance of data incoherence on the shared cache line A in a dual-core system. ❶ Initially, the shared memory has A with a value of 30. ❷ Core c_0 performs a read on A; hence, it obtains a local copy of A in its private cache. ❸ Afterwards, c_0 executes a write operation that updates this local copy to 50. ❹ When c_1 reads A, the shared memory responds with the old value of A, 30. This is because c_0 did not update the shared memory with the new value of A; thus, c_1 obtains a stale (incorrect) version of A.

A coherence protocol avoids data incoherence by deploying a set of rules to ensure that cores access the correct version of data at all times. Usually, the coherence protocol maintains data coherence at cache line granularity, which is a fixed size collection of data. A state machine implements these rules with a set of states representing read and write permissions on the cache line, and transitions between states denoting the activity of all cores on the shared data. The cache



(a) Incoherent sharing of data.

(b) MSI state diagram.

Figure 3.1: Cache coherence.

controller typically implements the coherence protocol. General purpose processors deploy different variants of coherence protocols. Most of them consist of three fundamental stable states, which establish the MSI protocol: *modified* (M), *shared* (S), and *invalid* (I) [10]. Figure 3.1b presents these states and the transitions amongst them. This state machine is implemented by the cache controller of each private cache. In addition, the shared cache controller implements a similar state machine to track the status of each block in the shared cache. A cache line in ***modified state*** means that the current core has written to it and did not propagate the updated data to the shared memory yet. Only one core can have a specific cache line in a modified state. A cache line in ***shared state*** means that the core has a valid, yet unmodified version of that line. One or more cores can have versions of the same cache line in shared state to allow for fast read accesses. A cache line in ***invalid state*** denotes the unavailability of that line in the cache or that its data is outdated. A cache controller changes the state of the cache line by observing the bus for coherence messages related to the same cache line by other cores, known as *bus snooping cache coherence* or receiving action messages from a centralized shared cache controller, known as *directory-based cache coherence*. In this thesis, we focus on bus snooping cache coherence as it is typically implemented in multi-core platforms with a small number of cores, which is the case in current real-time systems. For example, bus snooping is adopted in ARM chips such as [64]. For bus snooping protocols, we distinguish between two types of messages: *coherence messages* (Definition 3.1) and *data messages* (Definition 3.2).

Definition 3.1. ***Coherence messages***. *Coherence messages are messages that represent an action corresponding to a core's activity on the cache line.*

Definition 3.2. ***Data messages***. *Data messages are messages that represent data sent or received by a core as a consequence of a core's activity.*

If a core requests a cache line A for reading, it issues a GetS(A) message. If it requests A for writing, it issues a GetM(A) message. If A is modified and evicted, then the core issues a PutM(A) message, which triggers a write back to the shared memory. If the core has A in a shared state and wants to modify it, it issues an Upg(A) message. A core observes its own messages on the bus as well as messages by other cores. We refer to the former as Own, and to the latter as Other. For instance, in Figure 3.1b, when the core has a line in S state and observes its OwnUpg(), it moves to M state. In contrast, if it observes a GetM() issued by another core, OtherGetM(), it changes its state to I.

To fix the data incoherent scenario, we apply the MSI protocol for the example in Figure 3.1a, c_0 first issues a GetS(A) to obtain A for reading, and then issues an Upg(A) to modify it. When c_1 issues a GetS(A) to obtain A for reading, c_0 observes an OtherGetS(A) on the bus. Hence, it either sends the updated data to c_1 directly, or it writes back A to the shared memory and c_1 reads the new data from it. The former case is possible only if the architecture allows for direct cache-to-cache communication. In both cases, c_1 reads the updated data and c_0 moves to the shared state. The cache coherence protocol is responsible for orchestrating such communication and transfer of data.

3.3.1 Transient Cache Coherence States

Interconnecting cores with an atomic in-order bus prevents all other cores from utilizing the bus until the core that was granted access to the bus completes its request. Consequently, most modern systems implement non-atomic reordering buses for improved performance, where the stall time of one core waiting for data response can be used to service requests of other cores. One example is the QuickPath interconnect architecture [65] from Intel that is used for inter-processor communication. However, if the bus is not an atomic in-order one, then a memory request to a cache line may be intervened by other requests to the same cache line before it completes (non-atomicity), and coherence messages can be reordered by the bus (reordering bus). Hence, a set of transient states between stable states is required to capture events caused by intervening coherence messages due to a non-atomic bus architecture. Table 3.1 tabulates the transient states, which are necessary between the two stable states I and S. Similar transient states exist between other stable states [10]. We categorize the transient states imposed by a non-atomic reordering bus into two categories based on their semantics.

	Core events			Bus events						
	Load	Store	Replacement	OwnData	OwnGetM	OwnGetS	OwnPutM	OtherGetS	OtherGetM	OtherPutM
I	issue GetS/IS^d	issue GetM/IM^d	X	X	X	X	X			
S	hit	issue GetM/SM^{ad}	-/I		X	X	X		-/I	-/I
IS^d	stall	stall	stall	read/S	X	X	X		IS^{d}I	
IS^a	stall	stall	stall	X	X	read/S	X			X
IS^{ad}	stall	stall	stall	IS^a	X	IS^d	X			
IS^{d}I	stall	stall	stall	read/I	X	X	X			X

Table 3.1: Transient states between S and I in a conventional MSI protocol. *issue msg/state* means the core issues the message *msg* and move to state *state*. *-/state* indicates that there is no *msg* issued. Shaded cells represent the situations where no transition occurs, while cells marked with X denote impossible cases under correct operation [10].

1. **Transient states for coherence messages.** These transient states denote that a core is waiting for its own coherence message to be placed on the bus. A core's own coherence message may be delayed on the bus due to the presence of other messages on the bus. For instance, when a core c_i has a read request to an invalid line (I state in Table 3.1), it issues a GetS message. Because of the non-atomicity and reordering nature of the bus, c_i might receive its requested data before it observes its message on the bus as shown by [10]. In this case, c_i moves to a transient state (IS^a in Table 3.1) waiting for its OwnGetS() to appear on the bus before moving to the stable state, S.

2. **Transient states for data messages.** These states denote that a core is waiting for data either from a core that has the data in its private cache hierarchy or from the shared memory. For example, the IS^d transient state in Table 3.1 denotes that a core issued a GetS() and did not receive a data response yet.

3.4 System Model

We consider a multi-core system with N cores, $\{c_0,c_1,...,c_{N-1}\}$. Each core has a private cache, and all cores have access to a shared memory. This shared memory can be an on-chip LLC, an off-chip DRAM, or both. Tasks running on cores can have shared data. These tasks can belong to a parallel application that is distributed across cores, or different applications that communicate between each other. Cores can share the whole shared memory space similar to [3] or share part

of the memory space similar to [62]. We do not impose any restrictions on how the interference on the shared memory is resolved, whether it is the LLC or the DRAM. Furthermore, we do not require any special demands from the task scheduling mechanism. This allows one to integrate the proposed solution to current task scheduling techniques, and to various mechanisms that control accesses to shared memories in real-time systems. Cores share a common bus connecting private caches to the shared memory, where data transfers amongst private caches are only via the shared memory (no cache-to-cache transfer). Although some of the problems addressed in this chapter may also apply to systems supporting cache-to-cache transfer, those systems are not the focus of this thesis. The common bus transfers data and coherence messages between the shared memory and the private caches. For example, Figure 3.1a shows a two core setup where the cores are connected to each other and to the shared memory via a common bus. The common bus also transfers coherence messages deployed by the coherence protocol to ensure data correctness. The system deploys a predictable arbitration on the common bus. The proposed solution is independent of the core architecture, and the arbitration mechanism on the bus. However, the analysis and experiments we present in this chapter consider a system with in-order cores, and a TDM bus as the base arbitration scheme to derive WC latencies. TDM can be either time-conserving or non time-conserving. Time-conserving TDM grants the slot to the next core if the current core does not have pending requests, while in non-time conserving TDM such slot remains idle [3]. We use a TDM slot width that allows for one data transfer between shared memory and the private cache including the overhead of necessary coherence messages.

3.5 Sources of Unpredictability Due to Coherence

A cache coherence protocol ensures correctness of shared data across all cores in a multi-core platform. Nonetheless, careless adoption of a conventional coherence protocol into a real-time system may lead to unpredictable scenarios. As we show in this section, simply adopting a predictable arbiter in this case does not necessarily mean that tasks will have predictable latencies upon accessing the shared memory. This is because, as illustrated in Section 3.3, the latency suffered by one core accessing a shared line is dependent on the coherence state of that line in the private caches of other cores. A major contribution of this chapter is 1) to identify these unpredictable scenarios, and 2) to propose invariants to address them. It is worth mentioning that not all platforms necessarily suffer from all sources. Exact sources existing in platforms are implementation-dependent and not publicly-available. The proposed invariants are general design guidelines, which are independent of the adopted cache coherence protocol and the underlying platform architecture. Satisfying these invariants eliminates the identified unpredictable scenarios; thus, it leads to a predictable behavior. An arbiter manages accesses to the shared bus

such that at any time instance it exclusively grants the bus to a single core. A predictable arbiter guarantees that each core with a ready request will eventually access the bus; thus, it enables deriving latency bounds. Upon implementing a coherence protocol, a core initiates memory requests by exchanging coherence messages with other cores and the shared memory. Therefore, before investigating the potential sources of unpredictability, we extend the predictable bus arbiter with Invariant 3.1 such that it manages both data transfers and coherence messages.

Invariant 3.1. ***A predictable bus arbiter must manage coherence messages on the bus such that each core may issue a coherence request on the bus if and only if it is granted an access slot to the bus.***

Investigating the implications of a conventional coherence protocol on the WCET, we find that there are four major sources that can lead to unpredictable behavior. Figures 3.2–3.5 illustrate example scenarios for these sources (left side of each figure with the source of unpredictable behavior shaded in red). Figures 3.2–3.5 also illustrate how satisfying the proposed invariants prevents these actions and leads to a bounded memory latency (right side of each figure). Figures 3.2–3.5 consider a system with three cores, c_0, c_1, and c_2, and deploys a TDM arbitration amongst their requests to the common bus. If the request type is not specified whether it is a read or write, that means the scenario is agnostic to it. Each of Figures 3.2–3.5 separately defines the initial system state and the core under analysis for the corresponding scenario.

3.5.1 Source *1*: Inter-core Coherence Interference on Same Line

If a core requests a line that has been modified by another core, it has to wait for the modifying core to write back that line to the shared memory. Source 1 occurs when multiple cores request the same line, say A, where A is modified by another core. In Figure 3.2, initially, c_0 has a modified version of A in its private cache. ❶ c_2 issues a read request to A. Since c_0 has the modified version of A, it has to write back its data to the shared memory first. However, this is c_2's slot; thus, c_0 has to wait for its slot to perform the write back. ❷ c_0 writes back A to the shared memory in its slot. ❸ c_1 issues a write request to A. Since the shared memory has the updated version of A, c_1 is able to obtain A and modify it. ❹ c_2 reissues a read request to A. This time c_2 has to wait for c_1 to write back A. From c_2's perspective, slot ❹ is a repetition of ❶; c_2 reissues its request to A and waits for another core to write it back. Thus, this situation is repeatable and can result in unbounded memory latency. Although c_2 is granted access to the bus, it is unable to obtain the requested data due to the coherence interference.

Proposed solution. We avoid this problem by enforcing Invariant 3.2. Invariant 3.2 requires memory to service requests to the same cache line in their arrival order; thus, it guarantees that a line being requested by a core will not be invalidated before the core accesses it. Imposing

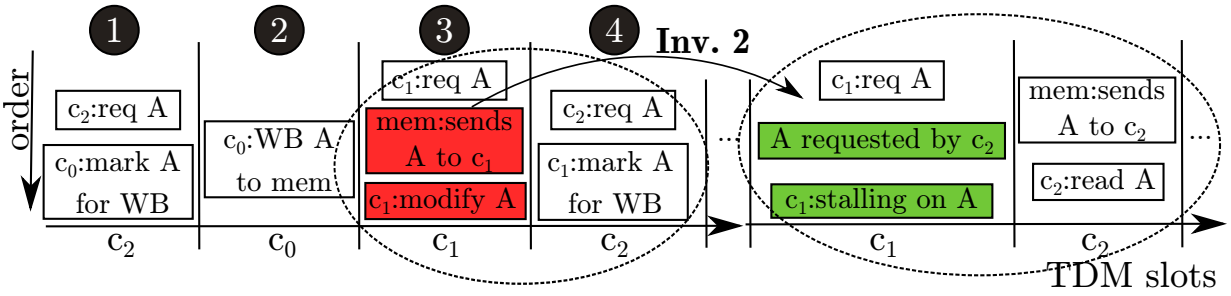


Figure 3.2: Unpredictability source 1: inter-core coherence interference on same line. Initially, c_0 modified A. c_2 is under analysis.

Invariant 3.2 in Figure 3.2, c_2's request to A arrives to the shared memory before c_1's request; therefore, c_1 has to wait for c_2 to execute its operation before it gains an access to A.

Invariant 3.2. ***The shared memory services requests to the same line in the order of their arrival to the shared memory.***

3.5.2 Source *2*: Inter-core Coherence Interference on Different Lines

This source of interference arises when a core has multiple pending lines to write back because other cores requested them. This is an indirect source of interference. Cores requesting access to different lines can interfere with each other because of the coherence protocol. For instance in Figure 3.3, c_0 has modified versions of lines A and B. The core under analysis is c_1. c_1 and c_2 issue requests to A ❶ and B ❷ during their corresponding slots. Accordingly, c_0 has to write back both A and B to the shared memory. Since c_0 is permitted to conduct one memory transfer in a slot, at ❸, it can write back only one line. If no predictable mechanism manages the write backs, c_0 can pick any pending one. At ❸, c_0 writes back B. Therefore, at ❹, c_1 is stalling on A. This situation can repeat indefinitely. While c_1 is waiting for A, c_2 can ask for another line, say C, which is also modified by c_0 and the same situation can repeat.

Proposed solution. We avoid this situation by enforcing Invariant 3.3. Invariant 3.3 imposes an order in servicing coherence messages from other cores (write backs, for example). The right side of Figure 3.3 deploys Invariant 3.3. Since the request to A arrives before that to B, c_0 has to write back A first then B; thus, a predictable behavior is guaranteed.

Invariant 3.3. ***A core responds to coherence requests in the order of their arrival to that core.***

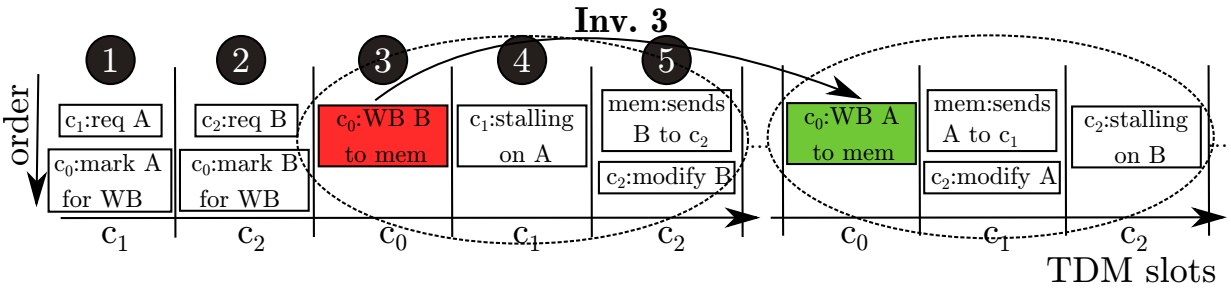


Figure 3.3: Unpredictability source 2: inter-core coherence interference on different lines. Initially, c_0 modified A and B. c_1 is under analysis.

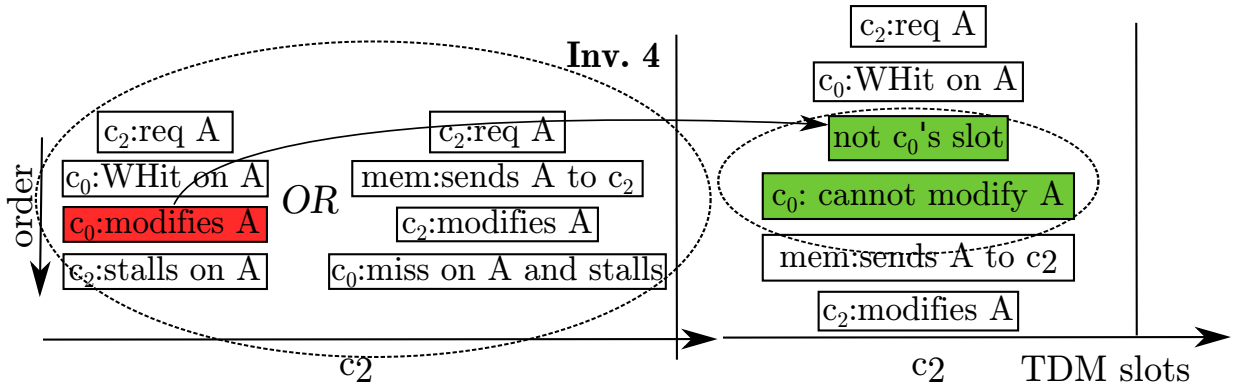
3.5.3 Source *3*: Inter-core Interference Due to Write Hits

This source is due to write hits in the private cache to non-modified lines. Since the predictable bus arbiter only controls accesses to the shared bus, a request that results in a hit in the private cache can proceed without waiting for the corresponding core slot. This yields two possible scenarios of interference as follows.

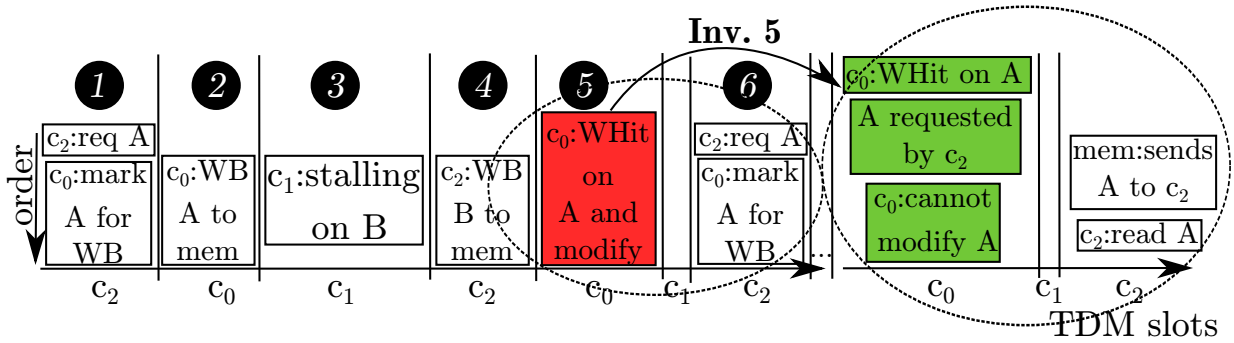
3.5.3.1 Source *3A*: inter-core interference due to write-hits to non-modified lines during another core's slot

Figure 3.4a exemplifies this scenario. c_0 has a version of A in its private cache that is not modified. During c_2's slot, c_2 issues a write request to A, while simultaneously c_0 has a write operation to A that results in a hit in its private cache. This creates a race with two possibilities. If c_0's write hit on A occurs first, c_2 has to wait until c_0 writes back A. On the other hand, if c_2's request appears on the bus first, c_0 has to invalidate its own local copy of A. Hence, c_0's request to A will be a miss and has to wait for c_2 to write back A before it gets another access to it. Assume that c_0's write hit occurs first and c_2 waits. After c_0 writes back A and during c_2's next slot, c_0 again has another write hit to A. Again, c_2 has to wait for c_0 to write back A. Consequently, this situation is repeatable and can starve c_2.

Proposed solution. We avoid this interference by enforcing Invariant 3.4. Invariant 3.4 stalls a write request by a core, which is a hit to a non-modified line until the arbiter grants an access slot to that core. Thereby, it avoids the aforementioned unpredictable consequences. It is worth noting that Invariant 3.4 aligns with Invariant 3.1 as follows. Invariant 3.1 mandates that a core can initiate coherence messages into the bus only when it is granted an access to it by the arbiter. Although a write hit to a non-modified line does not need data from the shared memory, it still



(a) Source $3A$: initially, c_0 read A. c_2 is under analysis.



(b) Source $3B$: initially, c_0 modified A, c_2 modified B, and c_1 requested B. c_2 is under analysis.

Figure 3.4: Unpredictability source 3: inter-core interference due to write hits.

needs to send coherence messages on the bus. This is necessary to invalidate local copies of the same line that other cores have in their private caches. Accordingly, a write hit to a non-modified line has to wait for a granted access by the arbiter. On maintaining Invariant 3.4 in Figure 3.4a, the following behavior is guaranteed. Since the current slot belongs to c_2, and c_0's request is a write hit to A, which is not modified, c_0 must wait for its slot to that request. On the other hand, c_2 issues its write request to A. Since no core has a modified version of A, c_2 obtains A from the shared memory and performs the write operation. c_0 invalidates its own local copy of A.

Invariant 3.4. ***A write request from c_i that is a hit to a non-modified line in c_i's private cache has to wait for the arbiter to grant c_i an access to the bus.***

3.5.3.2 Source $3B$: inter-core interference due to write hits to non-modified lines that are requested by another core

Invariant 3.4 resolves the race situation between a request generated by a core in its designated slot and write hits from other cores. Note that the write hits to non-modified lines can lead to another unpredictable situation that Invariant 3.4 does not manage. We illustrate this situation in Figure 3.4b. Initially, c_0 has a modified version of A, c_2 has a modified version of B, and c_1 has requested B. ❶ c_2 requests A to read; thus, in c_0's next slot, it updates the shared memory with the modified value of A ❷. Since c_2's request is a read, c_0 does not invalidate its local version of A. At ❹, c_2 has two pending actions: fetching A from memory, and writing back B to the memory in response to c_1's request. Assume that c_2 chooses to write back B. Therefore, its request to A waits for the next slot. At ❺, c_0 has a write hit to A. Consequently, since this is c_0's slot, it conforms with Invariant 3.4; thereby, it modifies A. At ❻, it has to reissue its request to A and wait for c_0 to write back A to memory again. From c_2's perspective, this situation is similar to the situation at ❶. Similarly, in next periods, after c_0 writes back A, it can have a write hit to A before c_2 receives it from the memory. Clearly, this situation is repeatable indefinitely, and creates unbounded memory latency for c_2.

Proposed solution. We avoid the unbounded latency by enforcing Invariant 3.5. Invariant 3.5 stalls a write request, which is a hit to a non-modified line until all waiting requests from previous slots are completed. Thereby, it avoids the aforementioned unpredictable consequences. Maintaining Invariant 3.5 in the right side of Figure 3.4b, the following behavior is guaranteed. During c_0's slot, it has a hit to A. Since A is non-modified by c_0 and is previously requested by c_2, the write-hit cannot be processed. Accordingly, c_2 obtains A from the shared memory in its next slot and performs its operation. c_0's request to A is issued afterwards in the corresponding slot.

Invariant 3.5. ***A write request from c_i that is a hit to a non-modified line, say A, in c_i's private cache has to wait until all waiting cores that previously requested A get an access to A.***

3.5.4 Source 4: Intra-core Coherence Interference

This interference is between two actions from the same core. The first action is its own pending request, while the second is its response to a request from another core. This response is for example, a write back to a line that this core has in a modified state. In Figure 3.5, c_0 has a modified version of A. ❶ c_2 requests A; thus, c_0 marks A to write back in its next slot. However, at ❷ c_0 is in its next slot and has its own request to B pending to issue to the bus. Thus, the write back of A waits for c_0's next slot. Similarly, at ❹, it has another pending request to another

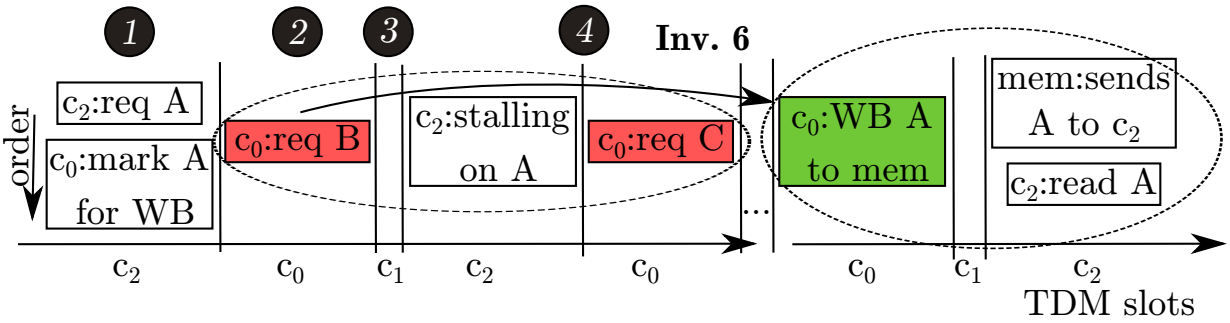


Figure 3.5: Unpredictability source 4: intra-core coherence interference. Initially, c_0 has modified A. c_2 is under analysis.

line, C. Accordingly, the write back of A by c_0 can indefinitely stall, which results in unbounded latency of c_2's request.

Proposed solution. We introduce Invariant 3.6 to resolve this intra-core interference problem predictably. Invariant 3.6 states that any predictable arbitration mechanism between coherence requests of a core and responses from the same core is sufficient to address the intra-core interference. Deciding the adequate arbitration depends on the application. Deploying Invariant 3.6 in Figure 3.5, the predictable arbitration mechanism will eventually allocate one c_0's slot to the write back operation of A. Thus, the memory latency of c_2's request is bounded.

Invariant 3.6. ***Each core has to deploy a predictable arbitration between its own generated requests and its responses to requests from other cores.***

3.6 PMSI: A Predictable Coherence Protocol

We show the effectiveness of the proposed invariants by applying them to the conventional MSI protocol. This results in the predictable PMSI protocol for multi-core real-time systems. To ensure these invariants are held, we propose architectural modifications and additional coherence states. The proposed architectural modifications satisfy Invariants 3.1 and 3.2 without any changes to the coherence protocol. However, Invariants 3.3–3.6 require modifications to both the architecture and the coherence protocol. This is because Invariants 3.3 and 3.6 regulate the write back operation of cache lines. Since a core has to wait for a designated write back slot to write back a cache line A, it has to maintain A in a transient state to indicate that A is waiting for write back. Similarly, Invariants 3.4 and 3.5 regulate the write hit operation to non-modified lines. A

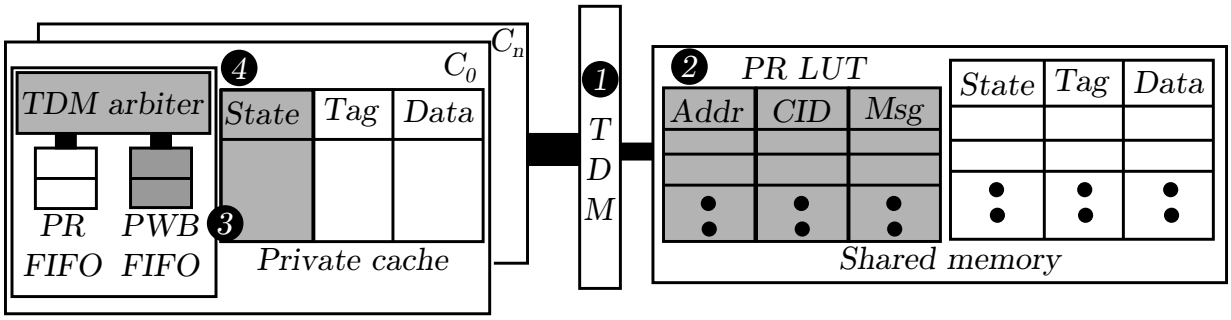


Figure 3.6: Architectural changes necessary for PMSI.

core has to wait for a designated slot to perform the write hit operation to a cache line, say B. Accordingly, it has to maintain B in a transient state indicating that it has a pending write to B.

3.6.1 Architectural Modifications

Figure 3.6 depicts a multi-core system with a private cache for each core and a shared memory connected to all cores via a shared bus. A TDM bus arbiter manages accesses to the shared memory. The proposed architecture changes are highlighted in grey. In our four-core evaluation system, the storage overhead is less than 128 bytes.

❶ The TDM arbiter manages the coherence requests such that each core can issue a coherence request message only when it is granted an access to the bus. This satisfies Invariant 3.1.

❷ The shared memory uses a FIFO arbitration between requests to the same cache line. We implement this arbitration using a look-up table (LUT) to queue pending requests (PR), denoted as PR LUT in Figure 3.6. Each entry consists of the address of the requested line, the id of the requesting core, and the coherence message. The PR LUT queues requests by the order of their arrival. When the memory has the updated data of a cache line, it looks for the first pending request for that line and services it first. This satisfies Invariant 3.2.

❸ Each core buffers the pending write back responses in a FIFO queue, which Figure 3.6 denotes as the pending write back (PWB) FIFO. This modification cooperates with the proposed transient states to satisfy Invariant 3.3.

❹ Each core deploys a time-conserving TDM arbitration between the PR FIFO and the PWB FIFO. This arbitration along with the proposed transient states comply with Invariant 3.6.

	Core events			Bus events						
	Load	Store	Replacement	OwnData	OwnUpg	OwnPutM	OtherGetS	OtherGetM	OtherUpg	OtherPutM
I	issue GetS/IS^d	issue GetM/IM^d	X	X		X				
S	hit	issue Upg/**SM^w**	X		X	X		I	I	X
M	hit	hit	issue PutM/**MI^{wb}**	X	X	X	issue PutM/**MS^{wb}**	issue PutM/**MI^{wb}**	X	X
IS^d	X	X	X	read/S	X	X		IS^dI	IS^dI	
IM^d	X	X	X	write/M	X	X	IM^dS	IM^dI	X	
SM^w	**X**	**X**	**X**		**store/M**	X		**I**	**I**	**X**
MI^{wb}	**hit**	**hit**		**X**	**X**	send Data/**I**			**X**	**X**
MS^{wb}	**hit**	**hit**	**MI^{wb}**	**X**	**X**	send Data/**S**		**MI^{wb}**	**X**	**X**
IM^dI	X	X	X	write/**MI^{wb}**	X	X			X	
IS^dI	X	X	X	read/I	X	X			X	
IM^dS	X	X	X	write/**MS^{wb}**	X	X		IM^dI	X	

Table 3.2: Private memory states for PMSI. *issue msg/state* means the core issues the message *msg* and move to state *state*. A core issues a *load/store* request. Once the cache line is available, the core *reads/writes* it. A core needs to issue a *replacement* to write back a dirty block before eviction. Changes to conventional MSI are in bold red.

These architectural changes, along with the coherence protocol changes, also satisfy Invariants 3.4 and 3.5 as follows. A core, say c_i, that has a write-hit to a non-modified line, say A, has to initiate a coherence message on the bus, known as an Upg(). With change ❶, the TDM bus arbiter does not allow this Upg() message on the bus unless it is the TDM slot of the initiating core. In consequence, the write-hit to A is postponed to the c_i's next slot, which implements Invariant 3.4. Assume that during c_i's next slot, there were one or more pending requests to A from other cores that arrived before c_i's request. According to Invariant 3.5, c_i's write hit to A has to wait until these pending requests are serviced. Recall that PR LUT ❷ queues pending requests. If the write-hit is to one of these lines, the arbiter does not elect the write-hit to execute during this slot. Accordingly, Invariant 3.5 is fulfilled.

3.6.2 Coherence Protocol Modifications

Table 3.2 shows all possible coherence states for a cache line and the transitions between these states for PMSI. We do not make changes to the coherence states for the shared memory, and hence it is not shown. Shaded cells represent the situations where no transition occurs, while

cells marked with "X" denote impossible cases under correct operation. Take for instance the case where c_i has a cache line A in state I. If c_i has a read operation to A, it issues an OwnGetS() message for A, and moves to state IS^d. On the other hand, if c_i observes an OtherGetM() message on the bus for cache line A that it has in state I, it does not make any change to A's state. Alternatively, c_i cannot have a replacement request for A, since A is originally invalid in its private cache. The three stable states I, S, and M have the same semantics as the conventional MSI protocol as explained in Section 3.3.

3.6.2.1 Removed transient states

Recall in Section 3.3, we categorized transient states into: 1) states that indicate the waiting for coherence messages to appear on the bus, and 2) states that indicate the waiting for data responses. For a real-time system, the first category is not needed. On deploying a predictable bus arbitration, once a core is granted access to the bus, no other core can issue a coherence message during that slot. This is assured by Invariant 3.1. Accordingly, during a core slot, its coherence messages are not disrupted by messages from other cores. For example, assume that c_i has a read request to a line A that is invalid in its private cache. During c_i's slot, it issues its OwnGetS() to the bus. Since c_i is the only core issuing coherence messages to the bus, it cannot receive its data before observing its OwnGetS() on the bus. Therefore, c_i atomically changes A's state from I to IS^d without the need to move to IS^a. By removing these transient states, PMSI has fewer total states and transitions compared to the conventional MSI protocol as in [10]. Thus, PMSI encounters no overhead in state encoding.

3.6.2.2 Unmodified transient states

In contrast, the second category that denotes the waiting for data response is required in a real-time system that deploys a predictable bus arbitration and does not allow for cache-to-cache transfers. This is because if c_i issues a request to a cache line that is modified by another core c_j, c_i must wait until c_j writes back that cache line to the shared memory. If cache-to-cache transfer is not allowed, this operation consumes multiple schedule slots. Accordingly, c_i has to move to a transient state indicating that it is waiting for a data response from the memory. In Table 3.2, these states include IS^d for a read request and IM^d for a write request. In addition, there are three other unmodified transient states in Table 3.2, IS^dI, IM^dI, and IM^dS. These states indicate that the core has to take an action after receiving the data and perform the operation. Figure 3.7 explains the necessity of these transient states with an illustrative scenario. In Figure 3.7, c_1 issues a read request to A ❸, which c_0 has modified ❷. c_1 changes A's state to IS^d waiting for c_0 to write back A to shared memory. Before c_1 receives the data, c_2 requests A to modify ❹. According to

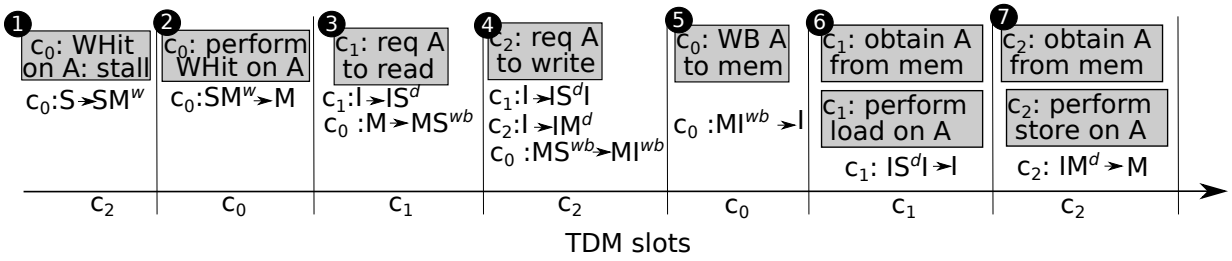


Figure 3.7: Transient states example; grey boxes are events, and arrows are state transitions. Initially, c_0 has A in S.

Invariant 3.2, the memory services c_1's request to A before c_2's request. However, c_1 has to store the information that there is a pending coherence message to A that it has to respond to once it completes its operation to A. This information is preserved by the transient states. For instance, during c_2's slot ❹, c_1 observes OtherGetM(A); thus, it has to move to IS^{d}I state to indicate that upon receiving the data and conducting the read operation, it has to invalidate A as c_2 will modify it ❻.

3.6.2.3 Proposed new states

We propose three additional transient states that are necessary to guarantee that invariants are upheld. Table 3.3 tabulates the proposed states along with their semantics. States MI^{wb} and MS^{wb} manage the write back operation. This is crucial for achieving predictability for any request to a modified cache line. For instance, in Figure 3.7, during c_1's slot, c_0 observes c_2's read request to A, which c_0 has modified ❹. Therefore, it marks A to be written back in the next slot by moving it to the MS^{wb} state. This indicates that c_0, in the next designated slot, will write back A to the memory and change its local copy of A to S state. Before c_0 writes back A to shared memory, it observes c_2's modify request to A ❹. As a consequence, it updates its A's state to MI^{wb}, which indicates that c_0's has to invalidate A once it performs the write back operation ❺. State SM^w is necessary to handle write hits to non-modified lines predictably. For example, in Figure 3.7, during c_2's slot ❶, c_0 has a write-hit to A, which it has in S state. To impose Invariant 3.4, c_0 has to postpone this operation to its next slot. Towards doing so, it updates its A's state to SM^w to preserve the information of the upgrade request to A. In its next slot, if no other core is pending on A (Invariant 3.5), c_0 issues its OwnUpg(A) on the bus, performs the write to A, and moves its A to the stable state M ❷.

Transient state	Initial state	Final state	Semantics
MI^{wb}	M	I	c_i has a line A in M state. Another core requested A to modify. MI^{wb} is necessary to reflect that c_i has to write back A in its next write back slot.
MS^{wb}	M	S	Similar to MI^{wb} except that the other core requested A to read.
SM^{w}	S	M	c_i has a write-hit to non-modified A in another core slot. c_i moves to SM^{w} until its allowable to perform the write-hit operation.

Table 3.3: Semantics of the proposed transient states to achieve a predictable behavior.

3.7 Latency Analysis

We derive the upper bound per-request latency that a core suffers when it attempts to access the shared memory. The considered system deploys the predictable MSI protocol proposed in Section 3.6 and a TDM bus arbitration amongst cores. We partition this latency into four components and compute the WC value of each of them. Definitions 3.3–3.7 formally define these latency components.

Definition 3.3. ***Arbitration latency.*** *The arbitration latency, $L^{arb}_{i,r}$, of a request number r generated by c_i, $req_{i,r}$, is measured from the time stamp of its issuance until it is granted access to the bus. $L^{arb}_{i,r}$ is due to requests from other cores scheduled before c_i.*

Definition 3.4. ***Access latency.*** *The access latency is the time required to transfer the requested data by c_i between the shared memory and the private cache of c_i. We assume that this data transfer takes a fixed latency, L^{acc}. This latency can be considered as the WC access latency of the shared memory.*

Definition 3.5. ***Coherence latency.*** *The coherence latency, $L^{coh}_{i,r}$, of a request $req_{i,r}$ generated by c_i is measured from the time stamp when c_i is granted access to the bus until it starts its data transfer. $L^{coh}_{i,r}$ occurs due to the rules enforced by the deployed coherence protocol to ensure data correctness.*

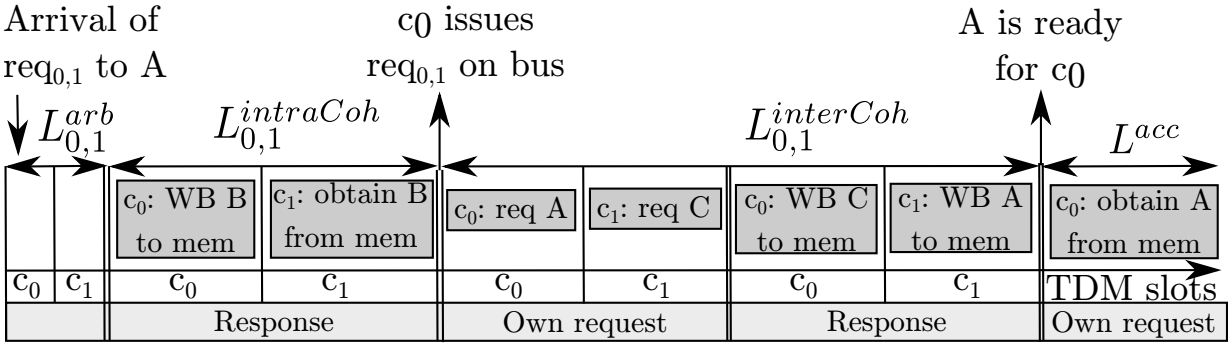


Figure 3.8: Different latency components. Initially, c_0 modified B and c_1 modified A.

We divide the coherence latency into two components: *inter-core* and *intra-core coherence latency*, which we denote receptively as $L^{interCoh}_{i,r}$ and $L^{intraCoh}_{i,r}$.

Definition 3.6. ***Inter-core coherence latency.*** *The inter-core coherence latency, $L^{interCoh}_{i,r}$, of a request $req_{i,r}$ generated by c_i is measured from the time stamp when $req_{i,r}$ is granted access to the bus until the data is ready by the shared memory for c_i to receive in c_i's slot. $req_{i,r}$ to a line* A *suffers inter-core coherence latency if another core has modified or requested* A *to modify before c_i issued its request.*

Definition 3.7. ***Intra-core coherence latency.*** *A request $req_{i,r}$ generated by c_i suffers intra-core coherence latency, $L^{intraCoh}_{i,r}$, if it has to wait until c_i issues a coherence response to an earlier request by another core. c_i is required to issue a coherence response when another core requests a line, say* B*, that c_i has in a modified state. Therefore, c_i needs to write back* B *to the shared memory.*

Figure 3.8 depicts the different latency components for a dual-core system with a TDM bus arbiter. c_0 issues $req_{0,1}$ to A one cycle after its slot has started; thus, it must wait for one TDM period before it accesses the bus in its next slot; hence, $L^{arb}_{0,1}$= 2 TDM slots. However, c_0's next slot is dedicated to write back responses. c_0 writes back B to shared memory since c_1 is pending for it. Consequently, $req_{0,1}$ must wait for another period before it gets an access to the bus. $L^{intraCoh}_{0,1}$ accounts for this delay and equals one TDM period. When c_0 issues $req_{0,1}$ to the bus, it turns out that c_1 has modified it. Therefore, c_0 has to wait for c_1 to update the shared memory with the new value of A. This waiting latency is the $L^{interCoh}_{0,1}$ and equals 2 TDM periods. Finally, the memory sends the data to c_0 and the transfer consumes L^{acc}, which is a single TDM slot.

Lemma 3.1. *The WC arbitration latency, WCL^{arb}_i, of any request generated by c_i occurs when c_i has to wait for the maximum possible number of requests generated by other cores before it can*

issue a request on the bus. For a system deploying conventional TDM bus arbitration, WCL_i^{arb} is calculated by Equation 3.1, where N is the number of cores and S is the TDM slot width in cycles.

$$WCL_i^{arb} = N \cdot S \tag{3.1}$$

Proof. Recall that the deployed TDM arbiter grants one slot to each core per period. Thus, the period equals to $N \cdot S$ cycles. The WC situation occurs when a request $req_{i,r}$ by c_i arrives one cycle after the start of c_i's slot. Consequently, $req_{i,r}$ has to wait for one TDM period until it is granted access by the bus, which equals to $N \cdot S$. □

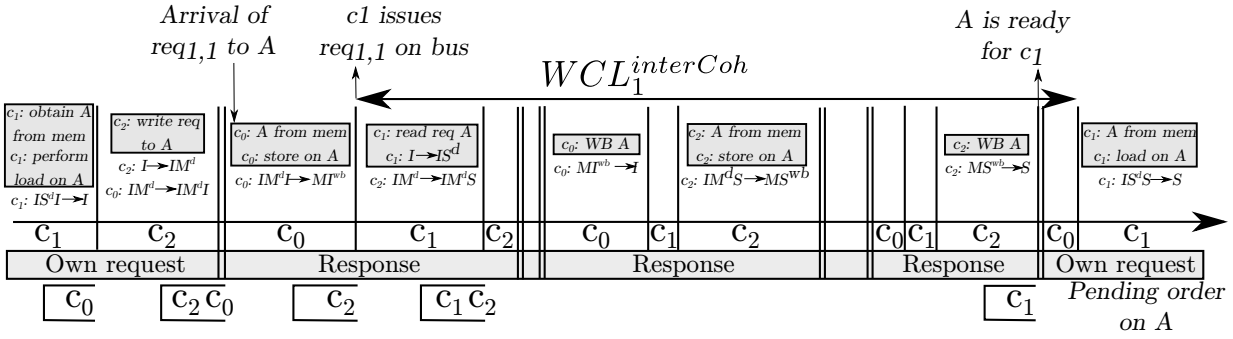
Lemma 3.2. *The WC inter-core coherence latency, $WCL_i^{interCoh}$, occurs when a core requests a line that has been previously modified or requested to modify by all other cores.*

Proof. As per Definition 3.6, $req_{i,r}$ to a line A suffers inter-core coherence latency if another core has modified or requested A to modify before c_i issued $req_{i,r}$. Thus, c_i has to wait until previously pending requests to A complete and the shared memory has the updated value of A before it gains an access to it. As a result, c_i suffers $WCL_i^{interCoh}$ when all other $N-1$ cores in the system requested to modify A before c_i issued its request. We prove this by contradiction. Assume that each core consumes T periods to obtain A, write to it, and update the shared memory with the new value. As a result, c_i must wait for $L_1 = (N-1) \cdot T$ periods before it accesses A. Now, assume that c_i suffers $L_2 = WCL_i^{interCoh}$ when N' cores requested to modify A before c_i issued its request, where $N' < N-1$. In this case, c_i must wait for $L_2 = N' \cdot T$ periods before it accesses A. Since $N' < N-1$, then $L_2 < L_1$. However, this contradicts the hypothesis that $L_2 = WCL_i^{interCoh}$. □

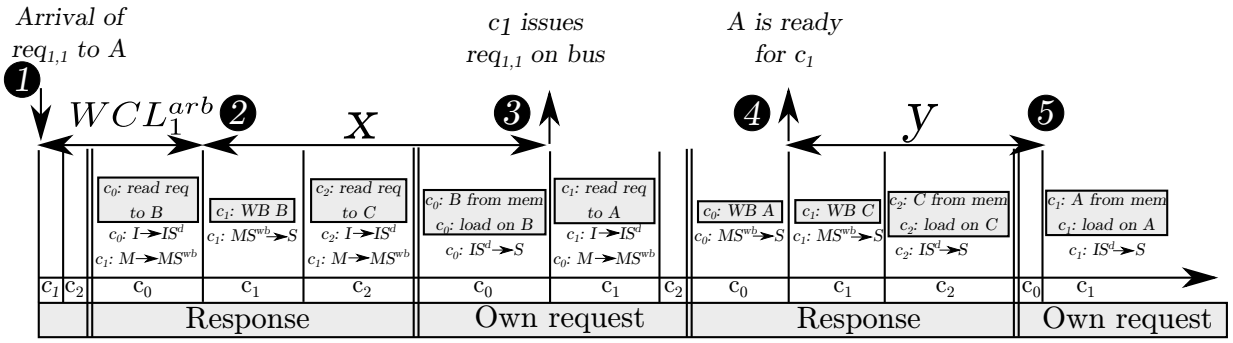
Lemma 3.3. *$WCL_i^{interCoh}$ is calculated by Equation 3.2.*

$$WCL_i^{interCoh} = 2N \cdot S \cdot (N-1) + \begin{cases} N \cdot S & N > 2 \\ 0 & N \leq 2 \end{cases} \tag{3.2}$$

Proof. From Lemma 3.2, c_i has to wait in WC for $N-1$ cores to obtain the line from the memory, perform the write operation, and finally update the shared memory with the new value. In WC, this procedure consumes two TDM periods for each other core, which leads to a total of $2(N-1)$ TDM periods. This accounts for the first component in Equation 3.2. Figure 3.9a shows the WC inter-core coherence latency for c_1 in a three-core system, where c_1 waits for 4 periods from the stamp of issuing the request to the bus until its data is ready to be sent by the memory. Moreover, if $N > 2$, when the shared memory has the updated version that is ready to send to c_i, c_i might have missed its slot in the current period. Therefore, it has to wait for an additional period to be



(a) WC inter-core coherence latency. Initially, c_0 and c_1 have pending requests on A with c_1 ordered first.



(b) WC intra-core coherence latency. $WCL_1^{intraCoh}$=$x+y$. Initially, c_0 and c_1 have pending requests on A with c_1 ordered first.

Figure 3.9: The latency bound on each interference component. Empty slots/periods do not have events that are related to c_1's latency.

able to receive A from the shared memory. In Figure 3.9a, the WC inter-core coherence latency of c_1 is 5 TDM periods, i.e. 15 slots. On the other hand, if $N \leq 2$, the core is guaranteed to have a slot in the same period as the data is ready at the memory. This is illustrated by Figure 3.8. This accounts for the second component in Equation 3.2. Recall that each period is $N \cdot S$ cycles. $WCL_i^{interCoh}$ is as calculated by Equation 3.2.

□

Lemma 3.4. *The WC intra-core coherence latency is calculated by Equation 3.3.*

$$WCL_i^{intraCoh} = \begin{cases} 2 \cdot N \cdot S & N > 2 \\ N \cdot S & N \leq 2 \end{cases} \tag{3.3}$$

Proof. 1. **Case of $N > 2$.** A request from c_i implies two actions from c_i. First, issuing the

request to the bus. Second, receiving the data from the shared memory. As a result, the worst-case intra-coherence latency occurs when each of these actions is delayed by write back responses that c_i has to conduct. Since the system deploys a time-conserving TDM between responses and own requests. Each action can encounter a maximum delay of one TDM period. Accordingly, the WC intra-coherence latency is two TDM periods or $2 \cdot N \cdot S$. Figure 3.9b delineates this situation. For the first action, although c_1 is granted a slot at time stamp ❷, it is a designated write back slot. Thus, c_1 does not issue its request until ❸, which is one TDM period later. For the second action, although the memory has the data ready for c_1 in its slot ❹, it is again a designated write back slot. Therefore, c_1 does not receive the data until ❺, which is one TDM period later.

2. **Case of $N \leq 2$.** Recall that cores are in-order such that each core can have at maximum one pending request at any instance. Hence, c_i cannot have two pending write back requests from the only other core in the system, c_j. In worst-case, c_i requests a line that is modified by c_j. Thus, it has to wait for two TDM periods because of inter-core coherence interference as per Lemma 3.3. In addition, c_i can have a worst-case arbitration latency of one TDM period as per Lemma 3.1. During this delay, which is three TDM periods at worst, c_i can have up to only one pending write back. This is because of the TDM arbitration between write backs and own requests. Figure 3.8 illustrates this situation.

□

Theorem 3.1. *The total WCL suffered by a core c_i issuing a request to a shared line A is calculated as:*

$$WCL_i^{tot} = (2 \cdot N^2 + 1) \cdot S + \begin{cases} 2 \cdot N \cdot S & N > 2 \\ 0 & N \leq 2 \end{cases} \tag{3.4}$$

Proof. Recall that L^{acc} is fixed and equals to the slot width, S. From Lemmas 3.1, 3.3, and 3.4 and since WCL_i^{tot}=WCL_i^{arb}+$WCL_i^{interCoh}$+$WCL_i^{intraCoh}$+L^{acc}, WCL_i^{tot} can be calculated by Equation 3.4. □

3.8 Evaluation

Architecture. We integrate PMSI into a full-system simulator called gem5 [52]. We use the Ruby memory model in gem5, which is a cycle-accurate model with a detailed implementation of cache coherence events. We use a multi-core architecture that consists of x86 cores running at 2GHz. The cores implement in-order pipelines, which we find are representative of cores used in the real-time domain. Each core has a private 16KB direct-mapped L1 cache, with its access

latency as 3 cycles. All cores share an 8-way set-associative 1MB LLC cache. Since the focus of this chapter is on coherence interference, we use a perfect LLC cache to avoid extra delays from accessing off-chip DRAM. Consequently, the access latency to the LLC is fixed, and equals to 50 cycles ($L^{acc} = 50$ cycles). The DRAM access overheads can be computed using the approach we propose in Chapter 4 or other approaches such as [60], and they are additive [29] to the latencies derived in this chapter. Both L1 and LLC have a cache line size of 64 bytes. The interconnect bus uses TDM arbitration amongst cores. The L1 cache controller uses time-conserving TDM arbitration between a core's own requests and its responses to other core requests. We do not run an operating system in the simulator, and hence, all memory addresses generated by the cores are physical memory addresses.

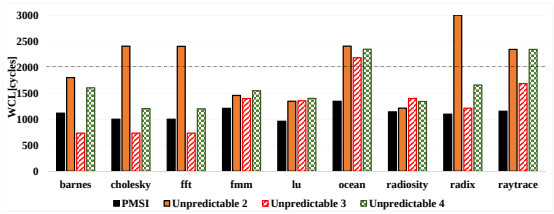
Workloads. We evaluate PMSI using the *SPLASH-2* [66] benchmark suite. In addition, we use synthetic workloads that we generated especially to stress the worst-case behaviour.

3.8.1 Verification

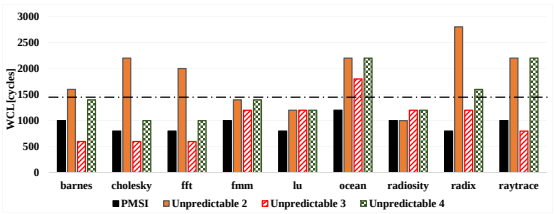
We verified the correctness of PMSI using various methods. 1) We used the Ruby Random Tester with gem5 [52] specifically to verify coherence protocols. We stressed PMSI with 10 million random requests. 2) We used carefully-crafted synthetic micro benchmarks to cover all possible transitions and states in PMSI. This also ensures the exhaustiveness of the identified unpredictability sources and corresponding invariants. Recall that PMSI only addresses the aforestated sources. All observed latencies conform to the bounds. If there was an unpredictability source that is not included in Section 3.5, it should lead to unpredictable behavior (i.e., observed latencies would exceed the bound) at one or more of the transitions, which we did not observe. 3) We executed the applications in the SPLASH-2 suite on gem5 using PMSI and they run to completion. Furthermore, we check data correctness by checking the output of each application.

3.8.2 Exp.1: Bounding the Memory Latency

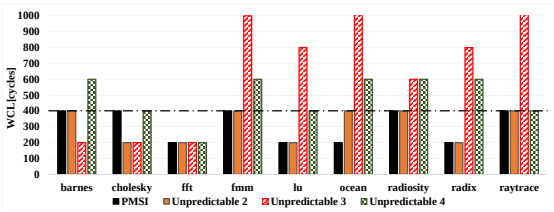
We study the effectiveness of PMSI to bound the delays resulting from coherence interference. We also study the effects of violating each one of the invariants on the memory latency. We use a 4-core system for our experiments. For SPLASH-2, we launch each SPLASH-2 application as four threads using four single-threaded cores, where only one application is used per experiment. Figure 3.10 depicts our findings. It shows the observed WC latencies for a) the total memory latency, b) the inter-coherence latency, and c) the intra-coherence latency. Since SPLASH-2 applications are optimized to minimize data sharing, they do not stress the coherence protocol. Therefore, to further stress the coherence protocol, we execute synthetic experiments using 9 synthetically-generated workloads: Synth1 to Synth9 in Figure 3.10.



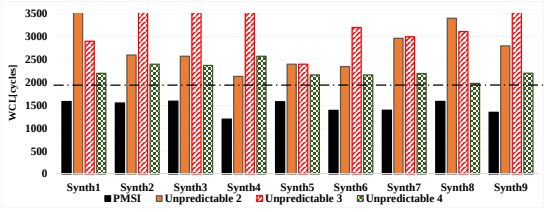
(a) Total memory latency for SPLASH suite.



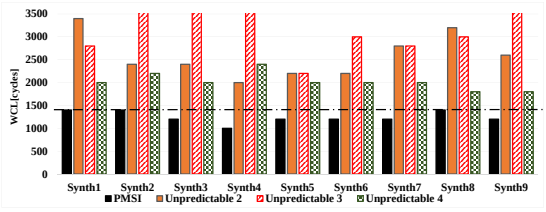
(b) Inter-core coherence latency for SPLASH.



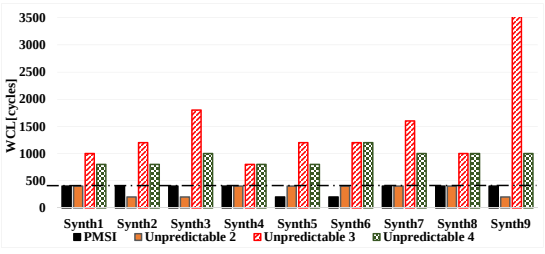
(c) Intra-core coherence latency for SPLASH.



(d) Total memory latency for the synthetic workloads.



(e) Inter-core coherence latency for the synthetic workloads.



(f) Intra-core coherence latency for the synthetic workloads.

Figure 3.10: WC latencies and the effect of unpredictability sources on them. Unpredictable i corresponds to source i in Section 3.5. Horizontal dotted line represents the analytical bound.

In each synthetic experiment, we simultaneously run four identical instances of one workload by assigning one instance on each core. These experiments represent the maximum possible sharing of data since each core generates the same sequence of memory requests. Figure 3.10 also illustrates the experimental WC latencies for these experiment. The WC arbitration latency for benchmarks in all experiments is $N \cdot S = 200$ cycles for $N = 4$ cores and slot $S = L^{acc} = 50$ cycles; hence, not shown.

Observations. 1) Figure 3.10 shows that for PMSI all the WC latencies are within their analytical bounds. 2) On the other hand, violating any of the invariants introduces a source of unpredictability, which results in exceeding those bounds. Moreover, for source 1, one of the cores is not able

to obtain an access to a block that it requests and the program never terminates. This is the reason that Figure 3.10 does not show Unpredictable 1. This shows that augmenting a conventional coherence protocol with a predictable arbiter does not guarantee predictability. 3) For a quad-core system, the latency suffered by a core due to coherence interference is $9\times$ more than the latency due to bus arbitration. The inter-core coherence interference solely contributes a latency up to $7\times$ of the arbitration latency, while the latency resulting from the intra-core coherence interference is double the arbitration latency. This provides evidence of the importance of considering the coherence latency when sharing data across multiple cores for real-time applications.

3.8.3 Exp.2: Comparing Performance with Conventional Protocols and Alternative Predictable Approaches

We compare the overhead caused by four approaches to handle data sharing in multi-core real-time systems: 1) not using private caches (*uncache-all*), 2) not caching the shared data (*uncache-shared*), 3) the proposed PMSI, 4) and mapping all tasks that share data to the same core (*single-core*). For the first three approaches, each application is distributed across four-cores. *uncache-shared* is an adaptation of the approach by [45, 46], but for data instead of instructions. *single-core* maps tasks with shared data to the same core to eliminate incoherence due to shared data, which adopts the idea of data-aware scheduling [47]. The overhead is calculated as the slowdown compared to the conventional MESI protocol. Figure 3.11 depicts our findings, where MSI and MESI are the conventional (unpredictable) protocols implemented as in [10].

Observations. 1) The *uncache-all* is useful when there is minimal amount of shared data being repeatedly accessed. From our experiments, we notice that data reuse is common in applications. This is the reason that *uncache-all* has the worst execution time for most applications with a geometric mean slowdown of $32.66\times$ compared to MESI. 2) Since private data does not cause any coherence interference, *uncache-shared* allows caching of only private data, while uncaching of all shared data. In Figure 3.11, *uncache-shared* has better performance than *uncache-all* for all applications with a geometric mean slowdown of $2.11\times$ Nonetheless, *uncache-shared* requires additional hardware and software modifications to distinguish and track cache lines with shared data, which are the same modifications required by [48]. 3) Mapping applications with shared data to the same core avoids data incoherence since these tasks share the same private cache. However, it prohibits parallel execution of these application. In consequence, for some applications (*fft*, *radix*, and *raytrace*), *single-core* achieves better performance compared to *uncache-shared*, while for other applications, it exhibits lower performance. This is dependent on several factors such as the memory-intensity of the application and the ratio of shared to non-shared data. Overall, *single-core* achieves a geometric slowdown of $2.67\times$. 4) Finally, PMSI achieves better performance compared to all other predictable approaches for all benchmarks, except for

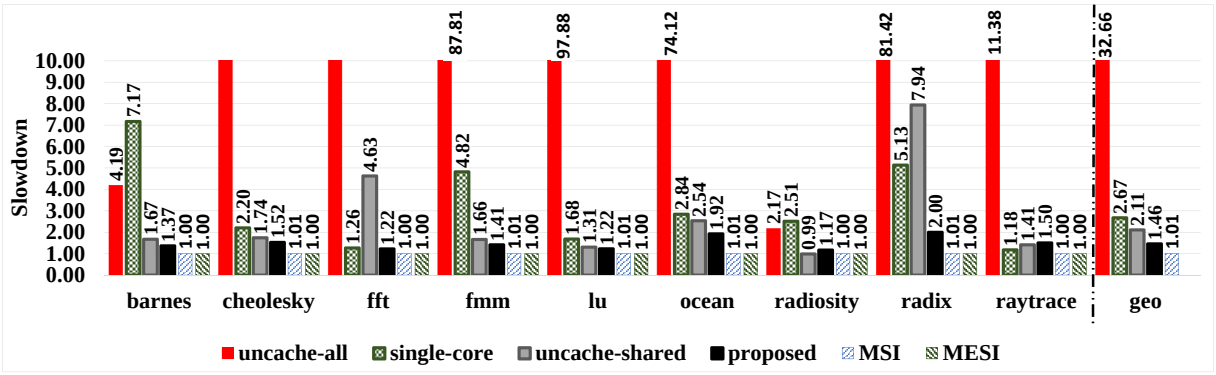


Figure 3.11: Execution time slowdown compared to MESI protocol.

radiosity and *raytrace*. PMSI achieves improved performance of up to $4\times$ the best competitive approach, *uncache-shared*, with a geometric mean slowdown of $1.46\times$ in performance compared to MESI. Upon analyzing *radiosity* and *raytrace*, we found that both do not show considerable reuse of the shared data. Shared lines in a core's private cache are often invalidated because of other cores before they are accessed again. Therefore, uncaching these cache lines achieves better performance. since the shared memory has the most updated value at all time instances; thus, cores do not suffer coherence interference.

3.8.4 Exp.3: Comparing to the Ideal Scenario

It is desirable to minimize the coherence interference, while allowing tasks to simultaneously access shared data. Optimally, the coherence interference equals zero. Although this is attainable only if all running tasks are independent so as they do not share data, it can be used as the ideal metric to compare different approaches to it.

Methodology. In this experiment, we study the slowdowns resulting from the proposed PMSI and the *uncache-shared* approach compared to the ideal case, *independent-tasks*. We stress both PMSI and *uncache-shared* by using our synthetic workloads similar to Exp.1. In *independent-tasks*, for each application, we simultaneously run three other applications in the three other cores, which do not share data with the current application. In *uncache-shared*, we simultaneously run four identical instances of each application, one instance on each core. Since these applications share their all memory data, *uncache-shared* and *uncache-all* are equivalent approaches; thus, we do not consider the *uncache-all* case. We delineate the results of these experiments in Figure 3.12.

Observations. 1) Since the data sharing is maximum for *uncache-shared*, all memory requests have to access the shared memory suffering from L^{acc} latency. Accordingly, *uncache-shared*

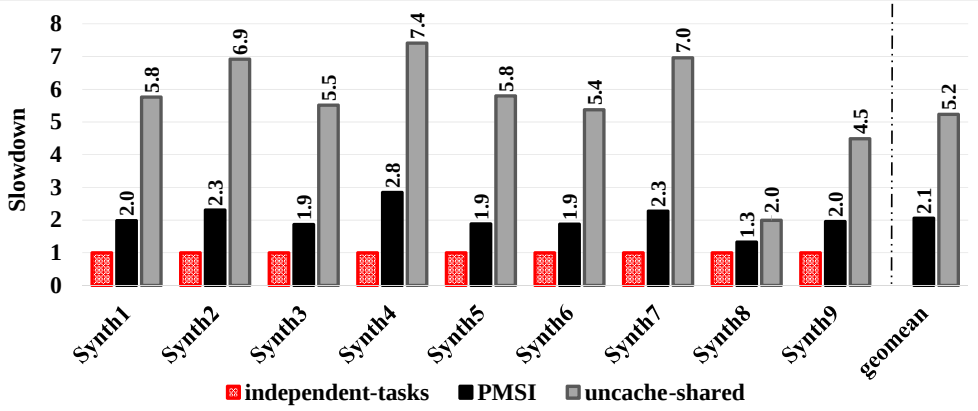


Figure 3.12: Slowdown in the execution time of different approaches compared to ideal scenario.

suffers from severe slowdowns compared to the ideal case, *independent-tasks*. In Figure 3.12, its execution time ranges from $2\times$ to $7.4\times$ compared to *independent-tasks*, with a geometric mean of $5.2\times$. 2) On the other hand, PMSI allows cores to cache the data (both private and shared) to their private caches, which improves the overall performance. PMSI's execution time ranges from $1.3\times$ to $2.8\times$ compared to *independent-tasks*, with a geometric mean of $2.1\times$. This illustrates the importance of deploying a coherence protocol to manage data sharing, while decreasing performance overheads.

3.8.5 Exp.4: Scalability

Methodology. We study the impact of each latency type upon increasing number of cores. We run one instance of the Synth1 workload on the core under consideration, c_0, and sweep the number of co-running cores from 0 to 7. Each co-running core also executes the Synth1 workload. We measure the experimental WC latencies of c_0 and delineate them in Figure 3.13.
Observations. Clearly, increasing the number of cores, the increasing rate in the inter-coherence latency is much larger than that of the arbitration and intra-coherence latencies. This aligns with the analysis in Section 3.7. The inter-coherence latency is a quadratic function, while both the arbitration and intra-coherence latencies are linear functions in the number of cores. Moreover, the increasing rate of the intra-coherence latency is double that of the arbitration-latency. Figure 3.13 shows that increasing the number of cores, the coherence-latency dominates the total memory latency. This emphasizes the importance of carefully considering the coherence latency impact on multi-core real-time systems upon allowing for simultaneous accesses to shared data.

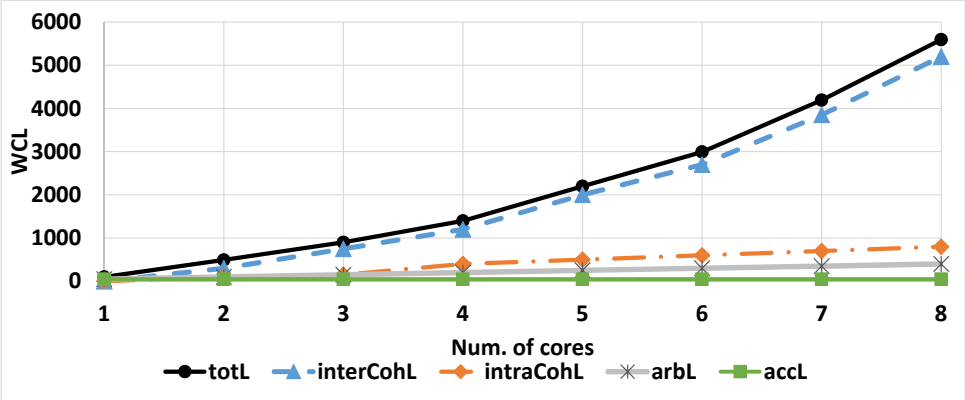


Figure 3.13: Latencies with different number of cores.

3.9 Summary

In this chapter, we pointed out possible sources of unpredictable behaviour on conventional coherence protocols. To address this unpredictability, we described a set of invariants. These invariants are general and can be applied to any coherence protocol. We showed how to deploy these invariants in the fundamental MSI protocol as an example. Towards this target, we proposed a set of novel transient states as well as minimal architecture requirements. In addition, we integrated the coherence effects on the latency analysis for the first time. We experimented using parallel applications from the SPLASH-2 suite as well as worst-case oriented synthetic workloads.

Chapter 4

PMC: A Requirement-aware DRAM Controller for Multi-core Mixed Criticality Systems

We propose a novel approach to schedule DRAM requests in MCS. This approach supports an arbitrary number of CLs by enabling the MCS designer to specify memory requirements per task. It retains locality within large-size requests to satisfy memory requirements of all tasks. To achieve this target, we introduce a compact harmonic work-conserving TDM scheduler, and a framework that constructs optimal schedules to manage requests to off-chip memory. We also present a static analysis that guarantees meeting requirements of all tasks. We compare the proposed controller against state-of-the-art MCs using both a case study and synthetic experiments.

4.1 Introduction

As aforementioned, MCS contain a mix of tasks with different criticalities. For example, HRT tasks are latency-critical and mandate strict assurances that their temporal requirements are never violated such that their worst-case latencies should always be no greater than their deadlines. Since a violation in temporal requirements of a HRT task may result in unacceptable loss of lives, a detailed WCET analysis of the task executions on the designated hardware platform is necessary. However, to compute tight WCET estimates, the hardware platforms must be predictable; thereby, leading itself to accurate WCET analysis. This means that speculative features such as out-of-order execution, complex cache hierarchies, and branch prediction are often eliminated [67]. Contrarily, SRT tasks can be considered as non latency-critical. They require a

minimum average-case performance and memory BW. To accomplish this, hardware platforms often use architectural features such as those disallowed for the purposes of predictability. Hence, the requirements of predictability for HRT tasks and average-case performance for SRT are in conflict. This poses an increasingly difficult challenge for designers of MCS.

One response to this challenge is utilizing temporal isolation [68–70]. Temporal isolation requires the designer to deploy the application such that resources used by tasks with strict temporal requirements are distinct. Heterogeneous multi-cores [71] with a combination of predictable and conventional processors offer an appealing hardware platform for deploying mixed criticality tasks. This is because HRT tasks can execute on predictable cores while SRT tasks execute on conventional cores. However, providing distinct off-chip memories to the cores is prohibitively costly, and researchers recognize that access to off-chip memories must be shared.

This has generated a considerable volume of research in the re-design of memory controllers (MCs) [59, 72–74] to control accesses to off-chip dynamic random-access memories (DRAMs). The key technique used in these works is to write back the data in the DRAM row buffer after each access. This is known as close-page policy, which ensures that every DRAM access consumes the same number of cycles and thereby achieves predictability. However, row locality between successive requests is not exploited. While this is apt for HRT tasks, SRT tasks experience significant bandwidth degradation. This makes such MCs ill-suited for SRT tasks. Goossens et al. [75] address this issue by keeping the data in the row buffer available for any further access within a designated time window. To exploit this for performance benefits, there must be multiple requests targeting the same row within a short time window. To address this limitation, Wu et al. [60] assign private DRAMs to each core to utilize the fact that accesses from the same core have a higher likelihood of exploiting row locality. Their approach prohibits sharing of data across the cores, and it requires assigning a DRAM bank per core, which may not be possible with a large number of cores. Three recent MCs [23, 76, 77] target MCS with critical and non-critical tasks. All the three MCs deploy a fixed-priority scheduling; thus, providing neither latency nor BW guarantees to any task other than the most-critical ones. While fixed-priority approach is suitable for dual-criticality systems, we find it ill-suited for systems with various mixed-critical tasks, where less-critical tasks may still require some guarantees [13]. As a result, we find that designers are still faced with the challenge of designing DRAM MCs that allow tasks with different memory requirements to share off-chip DRAMs in MCS while satisfying their respective temporal and bandwidth requirements.

4.1.1 Contributions

1. We directly address this challenge by proposing a novel requirement-aware approach to manage DRAM accesses in MCS. This is achieved by enabling the designer of MCS to assign

memory requirements (both BW and latency) per task/requestor[1]. To achieve this target, we introduce three components that together construct our approach. (1) A novel harmonic distributed time-division-multiplexing (TDM) scheduling scheme with low cost implementation adequate for MCS (Section 4.6). (2) A deployment framework to generate optimal schedules for the MC. The proposed framework is a tool to explore the trade-offs between requirements of different tasks to provide the optimal MC behaviour satisfying these requirements (Section 4.7.1). (3) Since MCS can deploy different sets of running tasks, we introduce PMC, a programmable memory controller that can be programmed with the optimal schedule at boot-time to meet varying requirements of different task sets in MCS (Section 4.5.1).

2. The proposed approach is based on the following novel observation. In MCS, SRT BW-sensitive requestors (such as multimedia processors, network processors and direct-memory access (DMA) processors) usually issue large-size memory requests. The observation we make is that if the locality within these large-size requests is exposed to the MC, we can minimize the WCL of HRT tasks, while satisfying the BW requirements of SRT tasks. To exploit the locality in large-size requests, we use a mixed-page policy that dynamically switches between close- and open-page policies based on the request size (Section 4.5.2). In addition, to bound the interference among different requestors, we provide an adaptive rate regulation mechanism (Section 4.5.5). The framework decides the optimal threshold of this rate-regulator to meet latency and BW requirements of all tasks.

3. Current DRAM modules are composed of a number of entities denoted as *banks*, where multiple banks can be combined in one group called *rank*. Based on the available DRAM module in the system as well as the characteristics of the set of running tasks, we introduce two architecture optimizations to PMC that further reduce DRAM latency and enhance its performance. (1) For systems with multi-rank DRAMs, requests can be interleaved across different ranks such that they are serviced in parallel. Rank interleaving is different than the rank switching mechanism proposed by [23, 59, 78] in that it parallelizes each request across available ranks instead of mapping consecutive requests to different ranks. We utilize this rank interleaving to reduce WCL, while increasing memory performance (Section 4.5.4). (2) To exploit the available bank parallelism, PMC interleaves each request across banks. Bank interleaving approach is followed by many real-time controllers such as [72, 74, 75, 79]. Interleaving requests with small transaction sizes across all available banks can result in unnecessary latency and bandwidth penalties [80]. Accordingly, we follow a similar approach to [79] and extend PMC to support interleaving across dynamic number of banks based on the issued transaction size (Section 4.5.3).

[1] Assuming a core affinity where each task executes on a designated core, we use words *task* and *requestor* in this chapter interchangeably.

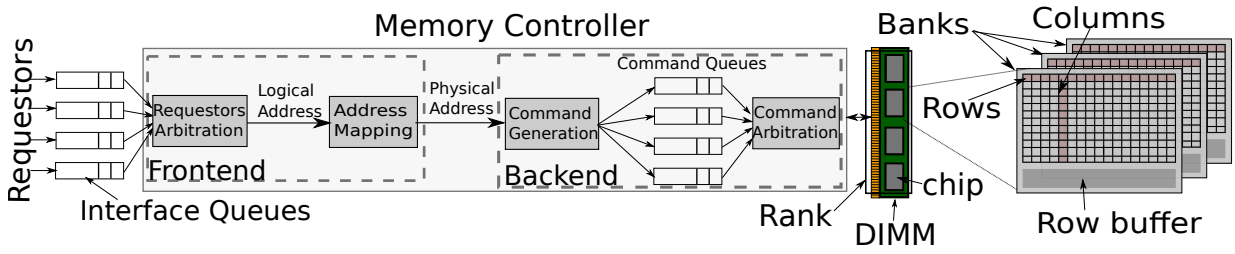


Figure 4.1: DRAM subsystem.

4. We present a static analysis for accesses to the DRAM managed by PMC in multi-core MCS to guarantee meeting the requirements of all requestors under all circumstances. The static analysis provides a different latency and BW bounds per requestor based on the generated optimal schedule (Section 4.7).

5. We provide a detailed comparison between the three different memory access policies: close-page policy [74], conservative-open page policy [75] and the mixed-page policy, which illustrates the strengths and scope of each one.

4.2 Background: Main Memory

As Figure 4.1 illustrates, a DRAM is organized in dual in-line memory modules (DIMMs), each DIMM consists of multiple DRAM chips. Each DRAM chip consists of memory cells arranged as *banks*. Cells in each bank are organized in *rows* and *columns*. A DRAM *rank* is a group of banks. Accesses to different ranks or banks can be interleaved to minimize the DRAM latency. We define the maximum amount of data that a DRAM can transfer when interleaving across all banks as the *memory granularity*, and it is equal to $BL \times n_{banks} \times CLW$, where BL is the burst length that can be 4B or 8B, n_{banks} is the number of banks the access interleaves across, and CLW is the column width in bytes. For multi-channel DRAMs, each *channel* has its own buses and consists of one or more ranks. Accesses to different channels, similar to ranks and banks, can be interleaved to reduce their access latency. On the other hand, accesses to different rows in the same bank suffer from *row conflicts* and encounter larger latencies. Data is transferred to/from the memory cells via sense amplifiers. These sense amplifiers work as a *row buffer* that caches the most-recently accessed row in each bank. A DRAM request consists of a type and an address. The type is either a read or a write. DRAM accesses are controlled

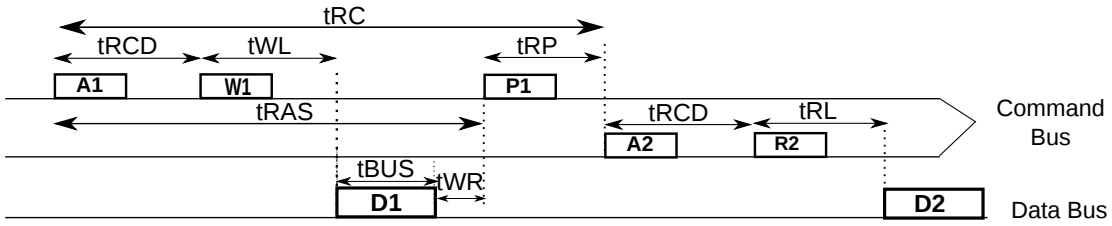


Figure 4.2: A write access followed by a write or read access targeting the same bank and rank for close-page policy.

by the MC, which translates the read/write requests into one or more of the following DRAM commands: ACTIVATE (A), READ (R), WRITE (W), PRECHARGE (P), and REFRESH (REF). A fetches the row from the memory cells to the sense amplifiers (row buffer). R (W) reads (writes) the required columns in the row buffer. P closes the activated row, and prepares the cell array for the next memory access by restoring the charge level of each DRAM cell in the row. Finally, REF activates and precharges DRAM rows to prevent charge leakage.

The DRAM JEDEC standard [11] imposes strict timing constraints on these commands (Table 4.1). All MC designs must satisfy these constraints to ensure correct DRAM behaviour. We use Figure 4.2 to illustrate the meaning of these constraints. It shows a write access followed by a write or read access targeting the same bank and rank. In Figure 4.2, $tRCD$ cycles are required between A1 and W1, and between A2 and R2. tWL cycles are required between the issuance of W1 and the start of writing data to the DRAM. Contrarily, tRL cycles are required between the issuance of R2 and the start of reading data from the DRAM. Then, the data transfer takes $tBUS$ cycles. tWR cycles are necessary between the end of the data writing, and the P1 command. tRP cycles are required between P1 and A2. These are timing constraints set by the physical properties of the DRAM.

Typically, a MC implements an arbitration scheme, an address mapping, and a page policy. The arbitration scheme arbitrates amongst different requests. The address mapping translates request addresses into its 5 segments: channel (ch), rank (rnk), bank (bnk), row (rw), and column (cl). We refer the number of bits assigned to channel, rank, bank, row and column indices as CNW, RKW, BKW, RWW, and CLW, respectively. This makes the physical address PW=CNW+RKW+BKW+RWW+CLW bits. The page policy controls the liveness of the row in the row buffer.

Table 4.1: Important JEDEC timing constraints (DDR3-1333) [11].

Const.	Meaning	Cyc.
tRC	Minimum time between A commands to same bank.	33
$tCCD$	Column-to-column delay.	4
tRP	Row precharge time	9
$tBUS$	$\frac{\text{request size}}{\text{data bus size} \times 2}$: Time required to transfer a data burst.	4
$tRAS$	Minimum time between A command and P command.	24
tWL	Minimum time between W and the start of data transfer.	7
tRL	Minimum time between R and the start of data transfer.	9
$tRCD$	Minimum time between activating the row and accessing it.	9
$tFAW$	Four bank activation window in same rank.	20
$tRTRS$	Rank to Rank switching delay.	1
$tRTP$	Read to precharge delay.	5
$tWTR$	Write to read switching delay.	5
tWR	Write recovery delay.	10
$tREFI$	Refresh Period.	$7.8\mu s$
$tRFC$	Time required to refresh.	$160ns$
$RKtoRK$	($tBUS + tRTRS$): Rank switching delay.	
$RtoW$	($tRL + tBUS + tRTRS - tWL$): R to W delay.	
$WtoR\_B$	($tWL + tBUS + tWTR$): W to R in same rank delay.	
$WtoR\_RK$	($tWL + tBUS + tRTRS - tRL$): W to R in different ranks delay.	
$RtoP$	($tBUS + tRTP - tCCD$): R to P delay.	
$WtoP$	($tWL + tBUS + tWR$): R to P delay.	

4.2.1 Memory Page Policies

There are two main page policies for accessing DRAMs: *close-page* and *open-page*. These page policies manage the duration during which the data is available in the row buffer. Close-page policy writes back the data in the row buffer and flushes the row buffer after each request. Under close-page policy, each request will consist of an A, a CAS, and a P commands. Hence, every request takes the same amount of access time, which helps deriving predictable latencies. Open-page policy, on the other hand, leaves the data in the row buffer to allow future accesses for data within the buffer to be accessed faster than having to read the data from the memory cells into the

row buffer again. MCs deploying open-page policy keep the row open until a request to another row arrives or the refresh period is reached. This enables open-page policy to be faster than close-page in the average-case. The primary drawback of open-page policy is that requests have a larger worst-case latency (WCL). This WCL occurs when a request targets a different row than the opened row, which requires precharging the opened row before loading the requested row in the row buffer. For these reasons, MCs in high-performance architectures often use open-page policy [81], while predictable MCs typically prefer close-page policy [23, 59, 72–74].

4.3 Related Work

There are several efforts that propose predictable MCs [23, 59, 60, 72–76, 79, 82, 83]. Most of these efforts [23, 59, 72–74] use close-page policy. Hence, available locality in the row buffer (known as row locality) is not exploited for performance benefits. The solution proposed by Goossens et al. [82] presents a configurable architecture where the MC can be reconfigured with different time division multiplexing (TDM) schedules that satisfy new run-time requirements. Gomony et al. [83] propose an optimal mapping of requestors to channels for a multi-channel MC. However, the latter two solutions also deploy a close-page policy, and do not exploit row locality.

Wu et al. [60] utilize open-page policy; however, they require each core to be assigned its own private DRAM bank. This makes their approach inapplicable when there is shared data between cores or the number of cores is greater than the number of DRAM banks. Goossens et al. [75] offer a compromise with their proposal of conservative open-page policy. This policy exploits row locality for SRT requestors while maintaining tight WCL bounds on HRT requestors. The proposed MC in [75] retains the data in the row buffer for a specified time window. When a request targets the same row in the row buffer and arrives within this window, it takes advantage of the row locality . While this approach allows SRT tasks to leverage performance benefits from open-page, it does not reduce the WCL compared to close-page policy. Furthermore, the proposed policy depends on the arrival time of requests. As noted by Wu et al. [60], non-trivial applications deployed on multi-core systems often require the designer to make no assumptions on the arrival times of memory requests due to multiple requests arriving from various cores. Unlike [75], PMC proposed in Chapter 4 requires no assumption on the arrival time of memory requests. In addition, unlike [60], we allow for shared data across cores.

Li et al. [79] deploy a MC backend that dynamically schedules DRAM access commands and supports different transaction sizes. Based on the transaction size, the numbers of interleaved banks and data bursts are determined through a look-up table. The backend issues DRAM commands on a FCFS basis. The dynamic command scheduling approach is promising for mixed crit-

icality systems since it increases average-case performance for requests of SRT tasks. Though, requests from HRT tasks incur same WCL of close-page controllers. PMC is a complete frontend and backend controller that promotes a mixed-page policy to decrease the WCL of memory accesses.

Ecco et al. [84] reduce the data bus switching delay by employing a CAS reordering technique. They schedule CAS commands in rounds such that all commands in the same round have the same type (read or write). Among the A and P commands, they deploy a RR arbitration. In [85], Ecco et al. extend this memory controller to support multi-ranked DRAMs. If the DRAM has multiple ranks, they schedule same type of CAS in one rank, and then switch to another rank to decrease the rank switching overhead.

Krishnapillai et al. propose ROC [78], a rank-switching open-row controller that forces consecutive requests to access different ranks to avoid the read-to-write and write-to-read switching time on the data bus. It deploys a RR arbitration across ranks and across banks of the same rank. ROC is able to decrease the WCL compared to [79]. However, it is complex to implement since it has three levels of arbitration on the backend only. Unlike [23, 78], which are rank-switching MCs, PMC deploys rank interleaving. While [23, 78] forces consecutive memory requests to access different ranks to avoid data bus switching, PMC interleaves each request across ranks to decrease its latency leveraging parallelism. In addition, consecutive accesses will be mapped to different ranks to avoid bus switching similar to [23, 78].

Three recent efforts have introduced MCs for MCS [23, 76, 77]. Jalle et al. introduce DCmc [76], which uses open-page policy and divides banks into critical and non-critical banks. They assign critical banks to critical requestors and schedule them using round robin (RR); hence, they provide latency bound guarantees for critical requestors. On the other hand, they assign non-critical banks to non-critical requestors and schedule them using first ready-first come first serve (FR-FCFS) to increase average-case performance. Ecco et al. introduce MCMC [23]. MCMC uses multi-ranks and bank partitioning with close-page policy. It assigns each bank partition to a critical requestor and a number of non-critical requestors. Then, it assigns critical requestors higher priority to eliminate the interference from the non-critical requestors. MCMC requires bank partitioning, which may limit shared data across requestors similar to [60]. Kim et al. [77] implement bank-aware address mapping and command-level scheduling to accommodate both critical and non-critical tasks. Banks are shared between both types of tasks. The command-level scheduling prioritizes commands of critical tasks. If a command from a critical request arrives while a non-critical request is being serviced, they pre-empt the non-critical request. As a result, the non-critical request has to be reissued again; thus, it suffers from performance penalty. Additionally, the first command from the critical request has to wait until satisfying all timing constraints after the pre-empted non-critical command. As observed by [86], this increases the latency of the critical request. All of the three MCs [23, 76, 77] are dual-criticality with fixed-

priority scheduling. Critical-tasks always have higher priority; hence, non-critical tasks have neither performance nor latency guarantees. This is acceptable for systems deploying only two types of tasks. Nonetheless, we find those MCs ill-suited for systems with various mixed-critical tasks, where less-critical tasks may still require some guarantees.

In contrast, PMC does not always prioritize higher-critical tasks. Instead, it executes an optimized schedule that allows the system designer to specify different latency and bandwidth requirements for each requestor. The schedule provides each task with the amount of service that is only sufficient to meet its specified requirements, while not starving other tasks.

4.4 Arbitration Mechanisms

A variety of arbitration mechanisms have been deployed by researchers for shared resources in real-time systems. Examples include RR [74], harmonic RR (HRR) [34], harmonic weighted RR (HWRR) [1], and TDM [59, 82]. Although RR is simple and efficient to implement, it shares the resource equally among different requestors regardless their type; and hence, it does not suit MCS. Yoon et al. [34] propose HRR to address this problem by assigning different periods to different tasks. They use HRR to maximize system utilization and not to minimize WCL. TDM scheduling is able to provide different services to different requestor types. The traditional TDM scheme is to allocate all slots assigned to a requestor contiguously in the schedule. The MCs in [75, 82, 83] follow this approach. Nonetheless, contiguous assignment of TDM slots does not provide tight WCL as each requestor has to wait, in worst case, for all other requestors before it is granted an access. PMC avoids this drawback by utilizing an optimized harmonic distributed assignment of the TDM slots. This is further discussed in Section 4.6. A recent effort that explores the state-space of TDM slot assignment has been proposed in [36]. Akesson et al. propose the credit-control-static priority (CCSP) scheduler [72]. CCSP is a fixed priority scheduler. During each period, lower-priority tasks in worst case have to wait for all higher-priority tasks to finish their budgets before it can issue a single request. Accordingly, they may or may not meet their temporal requirements. This has the same disadvantage of the contiguous TDM, which we discuss in Section 4.6. Contrarily, we propose an optimized schedule that allocates slots amongst running tasks in a harmonic distributed bases that is requirement-aware.

Conventional MCs usually deploy a FR-FCFS scheduling scheme among memory requests. FR-FCFS arbitrates amongst requests based on two factors: readiness and age. It prioritizes ready requests over non-ready requests. Ready requests are requests targeting an already open row. For two requests of the same category (both are ready or non-ready), FR-FCFS schedules the older first. This is not suitable for real-time tasks with tight timing requirements, since a non-ready request from a HRT task may suffer from extremely high WCL. Kim et al. [28] partially

address this problem by bounding the number of consecutive requests to the open row using a predefined threshold. However, this threshold is static and requestor-agnostic. Consecutive requests to the open row can belong to different requestors. Therefore, it is neither requestor- nor requirement-aware. Major differences between PMC's policy and FR-FCFS are as follows. 1) PMC arbitration is not age-based, it is an optimized TDM schedule that is requirement-aware. 2) The rate-regulation threshold of the mixed-page policy is per requestor and is independent of the row buffer state.

4.5 PMC: The Proposed Solution

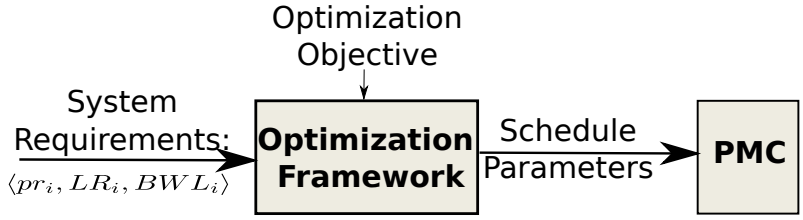
We define the input to the PMC to be memory requests from a set of m requestors, $R = \{r_1, r_2, ...r_m\}$. Each requestor executes a task until completion. Accordingly, a requestor is identified by the requirements of its running task. A requestor $r_i \in R$ is defined by the tuple: $\langle pr_i, LR_i, BWL_i \rangle$. pr_i is r_i's relative priority. It is an optional parameter that represents priorities of requestors, if they exist. More details about the role of pr_i are in Section 4.6. LR_i and BWL_i are the memory access latency and BW requirements of r_i, respectively. The derivation process of these requirements is out side the focus of this chapter. We propose a methodology to derive memory requirements for task sets in Chapter 2.

Figure 4.3a illustrates the proposed PMC framework. The framework takes as input the system requirements provided by the designer as the set of requestors R, and the optimization objective of the system determined by the designer.

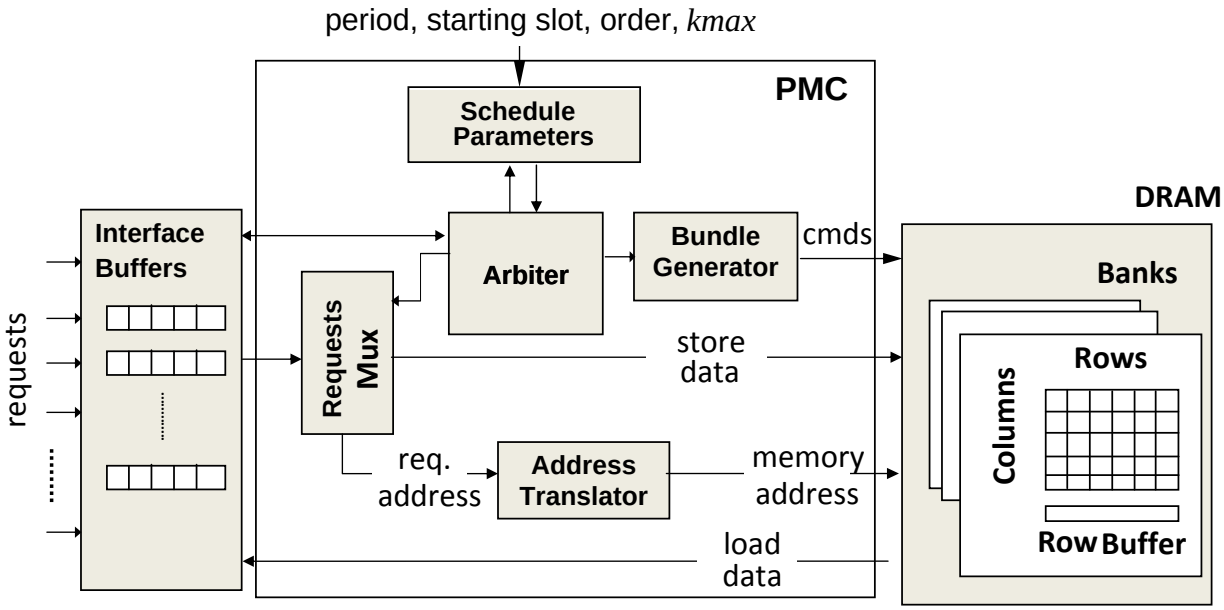
The designer can choose to optimize for the overall memory access latency, the access latency incurred by some of the requestors (HRT requestors for example), the overall BW provided by the DRAM, or the BW provided to some of the requestors (SRT requestors for example). The optimization framework determines the schedule parameters that satisfy system requirements and optimizes for the designer's target. These parameters are provided to the PMC at boot-time, which PMC uses to execute the arbitration schedule. We assume that these requirements does not change during running time. If a new set of tasks need to execute, the framework needs to rerun to determine the new optimal schedule parameters and provide them to PMC at boot-time.

4.5.1 PMC Architecture

We depict the proposed PMC architecture in Figure 4.3b. Requests from HRT and SRT tasks to the PMC are queued in the Interface Buffers. Each requestor is assigned a distinct interface buffer. Interface Buffers are typically part of the requestors architecture as load/store queues [87] or part



(a) Proposed methodology.



(b) Overview of PMC architecture.

Figure 4.3: PMC framework.

of the network-on-chip architecture known as transaction queues [88]. The Schedule Parameters block in Figure 4.3b is a look-up table to store the schedule parameters necessary to execute the schedule. The Arbiter executes the schedule identified by these parameters, and it also regulates the service rate provided to requests. Once a requestor is scheduled to access the DRAM by the Arbiter, the Request Selector retrieves the memory request from the Interface Buffers and supplies its address to the Address Translator. The Address Translator maps the physical address of the request to low-level addresses of the DRAM (rank, bank, row, and column addresses). The Bundle Generator generates low-level access commands to perform the access to the DRAM.

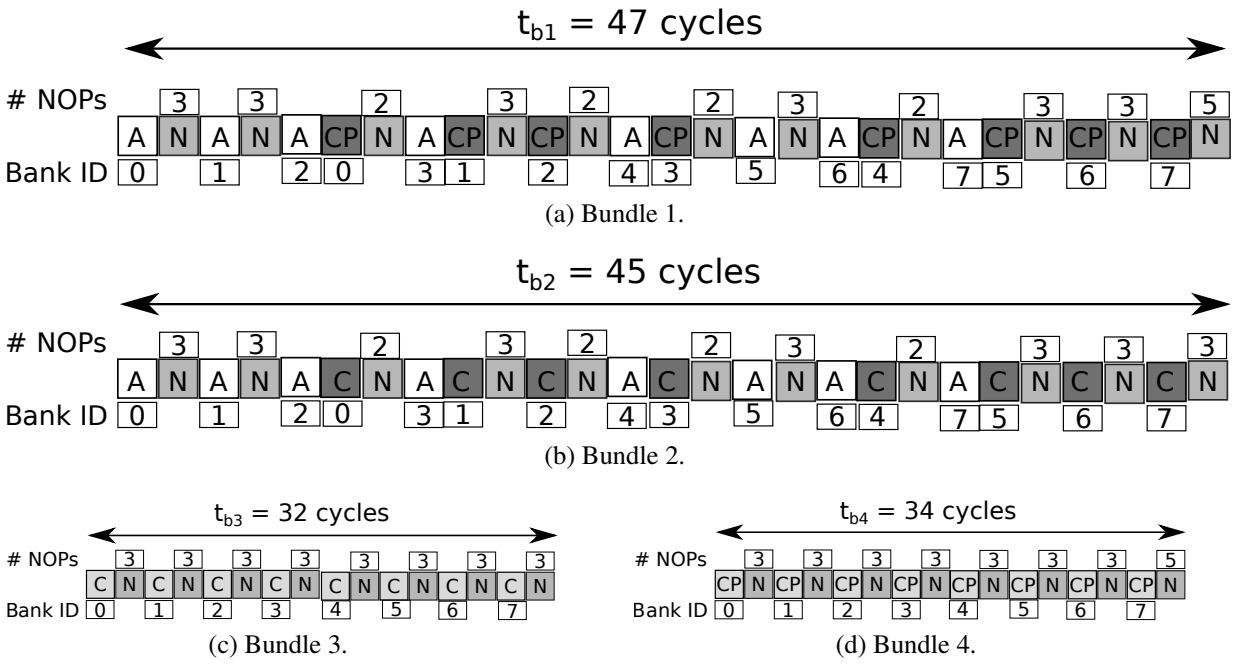


Figure 4.4: Command arrangements of the four bundles interleaving across 8 banks of DDR3-1333 for a write request. A: ACTIVATE command, C: CAS command, CP: CASp command, and N: NOP command.

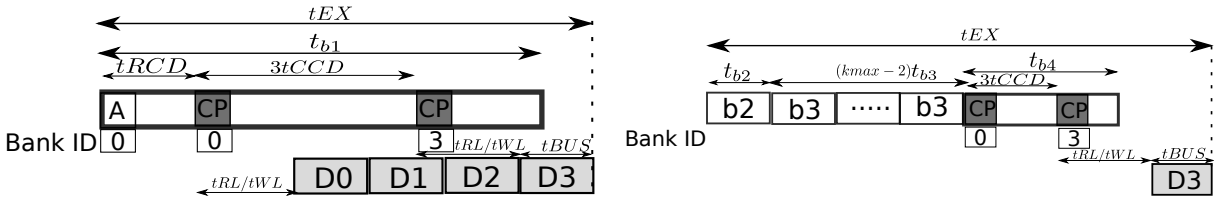
4.5.2 Formulating Bundles

We combine DRAM commands in statically defined groups with predictable behaviour that we call *bundles*. We construct four bundles of commands. Figure 4.4 describes the command arrangement for the four bundles in case of interleaving across eight banks. There are two numbers in the figure. The one at the bottom is the number of the bank being addressed, and the one at the top is the number of NOPs placed to satisfy the timing constraints. We use CASp to represent a CAS command with an automatic P command following it. Close-page policy uses CASp commands. Bundles 1 and 4 have CASp commands, which denote close-page policy, while bundles 2 and 3 use CAS commands, which denote open-page policy. Bundles 1 and 2 begin with an ACT command as they access the DRAM when the row is closed by a prior access. Conversely, bundles 3 and 4 begin with a CAS or a CASp command as they access the DRAM when their targeted row is already opened via prior bundles. A mix of these bundles promotes a run-time switching between close- and open-page policies. We construct the bundles to satisfy all constraints in worst case. For example, bundles 1 and 4 are padded with 5 NOPs to satisfy the

write-to-read switching constraints.

Although the command arrangements of the proposed bundles are similar to the groups proposed by [75], we use these bundles for a different target. The conservative open-page proposed by [75] leverages the command groups to increase the average-case performance, while maintaining the worst-case latency equivalent to the close-page policy. On the other hand, PMC leverages the command bundles to increase average-case performance and decrease the WCL compared to the close-page policy. This is achieved by exploiting the inherent locality in large-size requests. To achieve this, the Bundle Generator generates different bundle combinations for different requests based on their transaction sizes as follows. For a request with a transaction size that can be completed in one memory access, the Bundle Generator generates bundle 1 that implements close-page policy (Figure 4.5(a)). On the other hand, a request with a transaction size greater than the memory granularity is divided by PMC into multiple sub-requests, where each sub-request consists of a number of bundles. The number of sub-requests and the number of bundles granted to a sub-request are determined by the rate regulator as explained in Section 4.5.5. For a general sub-request, the Bundle Generator generates bundle 2 to open the targeted row, followed by a sequence of type 3 bundles deploying open-page policy accesses, and finally bundle 4 at the end to close the row (Figure 4.5 (b)). Therefore, rather than relying on the arrival time of requests, bundles 3 and 4 benefit from the row locality as they target an already open row in the row buffer. Consequently, as Figure 4.5 illustrates, the execution latency of each sub-request depends on its data size. We exploit this behaviour for tighter worst-case latency bounds while satisfying BW requirements (Section 4.6). We formally define the execution latency as follows.

Definition 4.1. ***Execution Latency.*** *The execution latency, t_{EXi}, of a sub-request of a request r_i is defined as the time elapsed from issuing the first command of that sub-request to the end of its data transfer from/to the DRAM. This time depends on the maximum number of consecutive*



(a) 1-bundle size sub-request.

(b) Multi-bundles size sub-request.

Figure 4.5: Bundles usage for the 4 bank interleaving case.

bundles granted to r_i ($kmax_i$) and is calculated as follows.

$$t_{EXi} = \begin{cases} tRCD + (n_{banks} - 1)tCCD + \chi(tFAW - 4tRRD) + tCL + tBUS, & \textit{if } kmax = 1 \\ t_{b_2} + (kmax_i - 2)t_{b_3} + (n_{banks} - 1)tCCD + tCL + tBUS, & \textit{if } kmax_i \geq 2 \end{cases}$$

Where:

$$\chi = \begin{cases} 0, & \textit{if } n_{banks} \leq 4 \\ 1, & \textit{Otherwise} \end{cases}$$

$$tCL = \begin{cases} tWL, & \textit{if } r_i \textit{ is a write request} \\ tRL, & \textit{if } r_i \textit{ is a read request} \end{cases}$$

Figure 4.5a illustrates t_{EX} for a single-bundle sub-request, while Figure 4.5b illustrates t_{EX} for a multi-bundle sub-request in case of interleaving across 4 banks.

4.5.3 Dynamic Bank Interleaving

Statically interleaving across all available banks simplifies the Bundle Generator , requires a small area overhead, and assist in deriving predictable latencies. Therefore, many predictable MCs follow this approach [3, 72–74]. However, this approach may result in transferring non-requested data; hence, non-utilized BW and/or unnecessarily larger memory latencies [73]. Equation 4.1 defines the percentage of non-utilized BW, BW_{NU}.

$$BW_{NU} = \frac{\text{transferred bytes} - \text{requested bytes}}{\text{transferred bytes}} \tag{4.1}$$

For example, if a requestor issues transactions of 32B, while the memory granularity is 128B, 75% of the BW delivered to this requestor is non-utilized, and this requestor's maximum utilization cannot exceed 0.25 regardless of the MC's efficiency. In addition, interleaving across 8 banks requires larger number of cycles than interleaving across only 2 banks. Hence, if those 32B transactions require only 2 banks, having a fixed 8-bank interleaving results in larger memory latency. If the transaction sizes of all requestors in the system is fixed and known in advance, the MC can interleave across the appropriate number of banks instead of fully interleaving across all banks. However, MCS have different requestor types with different transaction sizes. In addition, a single requestor may issue requests with different transaction sizes. Accordingly, a MC targeting MCS has to dynamically decide the number of banks to interleave across.

Table 4.2: Different bank interleaving for a single rank DDR3-1333. Bundle widths are in cycles.

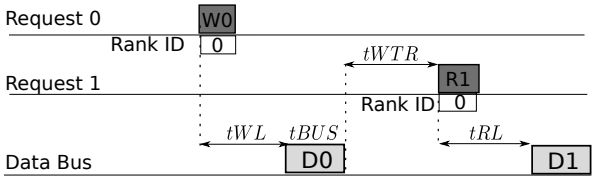
Banks	tb_1		tb_2	tb_3	tb_4		N Bytes
	R	W			R	W	
1	33	39	13	4	14	30	16
2	33	39	17	8	14	30	32
4	33	39	25	16	14	30	64
8	47	47	45	32	29	34	128

Bundle Formulation– To allow for smaller transaction sizes, we support interleaving across a dynamic number of banks based on the issued transaction size. Similar to Figure 4.4, where bundles interleave across 8 banks, we formulate bundles that interleave across any number of banks. Table 4.2 tabulates the number of cycles each bundle consumes across different number of banks. We claim that adding this dynamic interleaving support requires a minimal additional area overhead. This is due to two observations. First, requests with transaction sizes less than the memory granularity are 1-bundle size requests, thus, only bundle 1 needs to be stored. Second, memory transactions are usually 2^N bytes; thus, only a subset of the possible number of banks are practically needed (namely, 1, 2, 4 and 8 as Table 4.2 shows). Since we construct bundles to target the worst case, the values of tb_1 and tb_4 in Table 4.2 assume that consecutive requests are targeting same banks; hence, tRC and write-to-precharge constraints are considered.

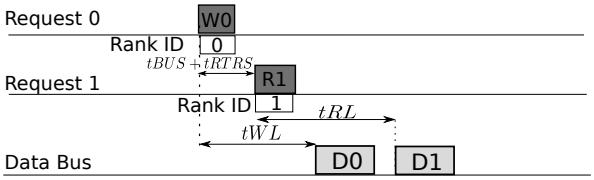
4.5.4 Rank Interleaving

The read-to-write and write-to-read switching times significantly increase the memory latency when successive requests to the same rank are of different type (a read followed by a write or vice versa). Since the bundle formulation in Figure 4.4 represents the worst-case of a write request, it considers the next request to be a read. Hence, the last CASp command of the bundle and the first CASp command of the next bundle must be separated by $tWL + tBUS + tWTR$; thus, we have the 5 NOPs at the end of the bundle to satisfy these constraints. Figure 4.6a demonstrates this situation. In contrast, each rank has its own data bus; thereby, no switching time is required between requests of different type.

Instead, there is a different constraint for successive accesses targeting different ranks: the rank-to-rank switching, $tRTRS$ as Figure 4.6b illustrates. $tRTRS$ is one to three cycles in different DDR modules, and is less than the read-to-write and write-to-read switching times. We promote a bundle formulation that leverages rank interleaving to avoid the switching latencies; thus, it decreases both average- and worst-case latency.



(a) Requests targeting same rank (latency = 29 cycles).



(b) Requests targeting different ranks (latency = 19 cycles).

Figure 4.6: A write followed by a read both targeting an open row.

Bundle Formulation– Figure 4.7 depicts one formulation example of bundles interleaving across two ranks. To avoid the data bus switching times, the first and last set of commands of each bundle access different ranks. In Figure 4.7, the time period between the CAS (or CASp) command targeting bank 4 and rank 1, and the one targeting bank 3 and rank 0 is 6 cycles to accommodate for $tBUS + tRTRS$ constraint between CAS commands targeting different ranks. For bundles 1 and 4, the rank interleaving avoids the data bus switching time between the last CASp command and the first CASp command in the next bundle; hence, provides less latency. For bundle 3, rank interleaving incurs more latency than the non interleaving case. This is because bundle 3 is always followed by either bundle 3 or bundle 4 of the same request. Accordingly, the non interleaving case in Figure 4.4c does not suffer from a bus switching time. On the other hand, in case of rank switching in Figure 4.7c, it suffers from the $tRTRS$ constraint. It is also important to highlight that Figures 4.4 and 4.7 shows interleaving across 8 banks. In case of interleaving across less number of banks, the timing constraints between the first CASp command in a bundle and the first A command in the next bundle subsumes the bus switching time. As a result, rank interleaving does not reduce the latency for this case. Figure 4.8 shows this situation for a single bank bundle case. $tWR+tRP+tRCD$ equals 28 cycles, while $tWTR$ is 5 cycles only. As a conclusion, the rank switching effectively reduces the latency for small size requests that interleave across 8 banks.

4.5.5 Arbitration Logic

The Arbiter in Figure 4.3b executes the schedule based on the schedule parameters to arbitrate accesses among requests. In addition, the Arbiter performs a rate regulation mechanism to prevent any single requestor from saturating available resources. For a requestor $r_i \in R$, a maximum number of bundles that can be serviced per access is defined as $kmax_i$. The Arbiter receives the request information (data size and requestor identifier) and computes the total number of bundles needed by the request (k_i). If $k_i > kmax_i$, the Arbiter splits the request into $\left\lceil \frac{k_i}{kmax_i} \right\rceil$

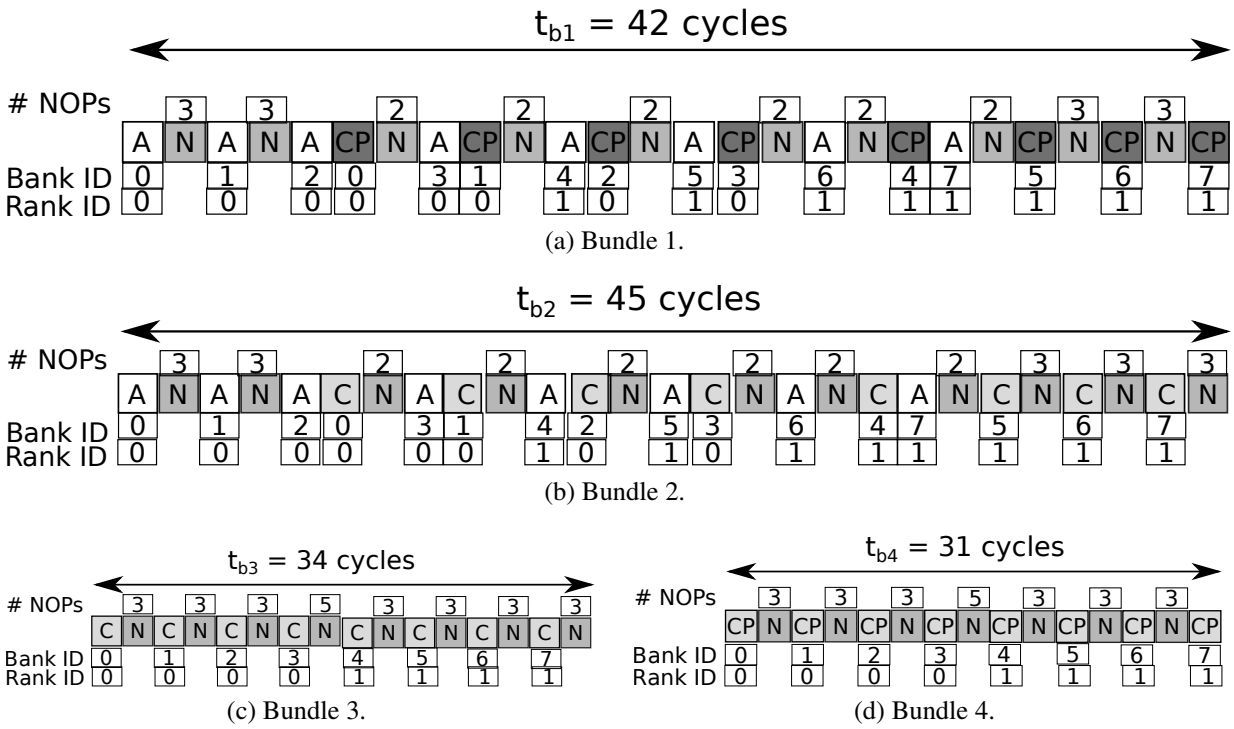


Figure 4.7: Command arrangements of the four bundles interleaving across 2 ranks and 4 banks per rank.

sub-request accesses. $kmax_i$ is calculated by the optimization framework for each requestor based on the system requirements. When a sub-request of data size RS_i bytes from requestor r_i is granted access to the DRAM, the Bundle Generator computes the number of bundles needed as $k_{subi} = \lceil \frac{RS_i}{BS} \rceil$, where $BS = BL \times n_{banks} \times DW$ denotes the bundle data size, which is equal to the memory granularity. BL is the burst length that can be 4 or 8, n_{banks} is the number of banks the access interleaves across, and DW is the data bus width in bytes (2B in our used DRAM). Hence, assuming $BL = 8$, BS is 128B in case of interleaving across all the eight banks of DDR3 and 64B when interleaving across four banks only.

4.6 Schedule Generation

There are two common types of TDM schedules: contiguous TDM and distributed TDM [89]. They are distinguished based on how the slots are assigned. Figure 4.9 shows an example of

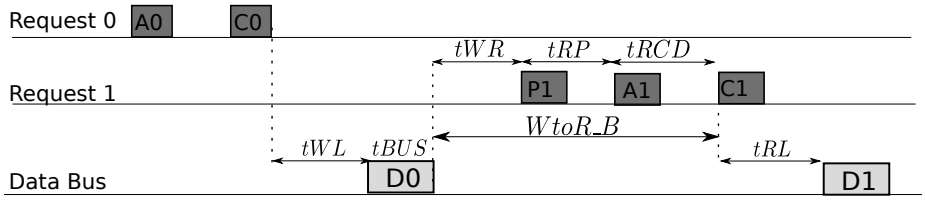


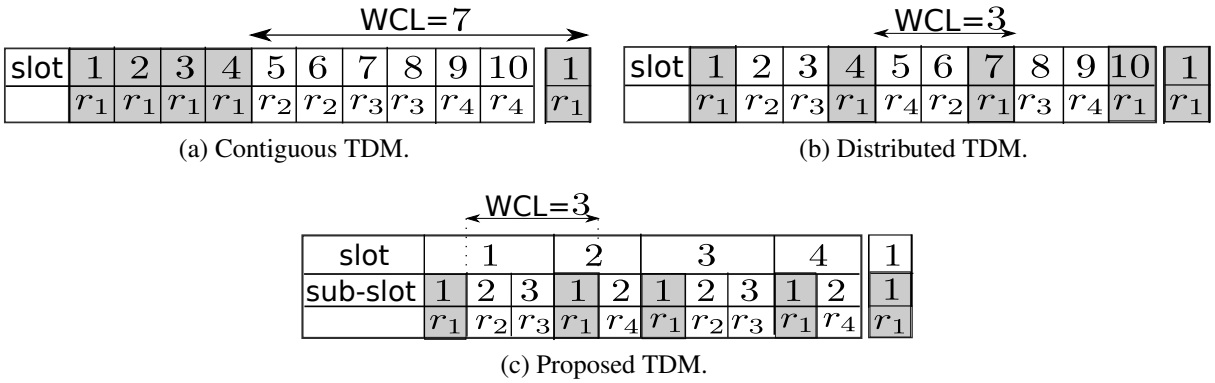
Figure 4.8: A write followed by a read for no bank interleaving (single bank bundle).

four requestors (r_1, r_2, r_3 and r_4) scheduled by contiguous (Figure 4.9a) and distributed TDM (Figure 4.9b), where r_1, r_2, r_3 and r_4 are assigned $4, 2, 2$ and 2 slots, respectively. Contiguous TDM assigns slots to each requestor in a consecutive fashion. In Figure 4.9a, for a total of 10 slots, the first four are assigned to r_1. Let the WCL be the time elapsed from the arrival of the request until it is completed. Then, the WCL of r_1 is 7 slots, which allows all other requests to access the resource before granting access to r_1. The advantage of contiguous TDM is that it is easy to implement with small area overhead. Basically, only the number of slots per requestor and the order of served requestors need to be stored. However, the downside of contiguous TDM is that the WCL of each requestor is larger compared to distributed TDM. For example, although r_1 gets 4 slots out of 10, its WCL is 7 slots.

In contrast, distributed TDM as shown in Figure 4.9b distributes the slots assigned to each requestor. Accordingly, the WCL of requestors in the distributed TDM schedule is less than that of the contiguous TDM. For example, r_1 in Figure 4.9b gets assigned once every two slots. This results in a WCL of 3 slots. Nonetheless, distributed TDM is more difficult to implement compared to the contiguous TDM as, in general, the whole schedule must be stored. This is because it is hard to equally distribute slots of each a requestor in the schedule. This is due to two challenges. First, the number of allocated slots to a requestor are not necessarily evenly divisible by the total number of slots in the schedule, known as the frame size. For instance, r_1 in Figure 4.9b needs 4 slots while the frame size is 10. Second, two requestors may require to be assigned the same slot. In consequence, the whole distributed TDM schedule has to be stored which may require large area overhead.

4.6.1 Proposed Implementation

To overcome the above-mentioned limitations, we propose a novel method to implement a distributed TDM schedule by applying two modifications. First, we set the frame size as a variable, which the framework determines its value based on the system requirements. Hence, we set the framework constraints such that the number of slots assigned to each requestor is divisible by the frame size. Consequently, we avoid the first challenge. Second, we divide each TDM slots



(a) Contiguous TDM.

(b) Distributed TDM.

(c) Proposed TDM.

Figure 4.9: TDM scheduling mechanisms.

into *sub-slots* such that the framework can assign multiple requestors to the same slot one after the other in successive sub-slots. As a result, unlike conventional TDM schedules, the slot width is intentionally variable. The optimization framework also determines the order of requestors within a slot by taking into account the relative priorities of requestors. This addresses the second challenge. Using our approach, we store the following parameters for each request: the period, the starting slot and the order in the slot. We explain these parameters in details in Section 4.6.2 For example, for r_1 in Figure 4.9c, the period is 1, the starting slot is 1 and the order is 1, which means that r_1 occupies the first sub-slot in each slot. Figure 4.9c shows that we have four slots, and these four slots have multiple requests such that each requestor posses a number of sub-slots. The details of the slot assignment is discussed in Section 4.6.3.

The proposed scheduler is work-conserving. A slot will not be idle unless no requestor has a ready request at this slot. In non work-conserving TDM scheduling, the time slot assigned to a requestor remains idle if there are no requests from this particular requestor. This conservative approach may be suitable for composable systems to force the latency to be equal to the WCL. However, it reduces system utilization and increases average latency. On the other hand, the proposed schedule grants access to the next scheduled requestor in case there are no requests from the current requestor. This is important to increase the utilization of shared resources, and improve the average-case performance. In the remaining of this section, we explain the details of the schedule and the schedule parameters. Based on these parameters, we compute the WCL bounds for any request accessing the DRAM using timing analysis in Section 4.7. For clarity, we tabulate all the terms used in the remaining of the chapter in Table 4.3, and accompany them with their explanations.

4.6.2 Schedule parameters

The fact that MCS execute tasks with different temporal and bandwidth demands raises the importance of having a programmable memory controller. Most existing predictable DRAM memory controllers employ static schedules (examples include [59, 72–75]); hence, they lack the ability to meet these demands. In PMC, schedule parameters are loaded at boot-time to the Schedule Parameters look-up table, which allows PMC to execute a different schedule that suits the running set of applications.

Area Overhead. The assignment of slots to requestors is harmonic. This increases the slot utilization, which we discuss in details in Section 4.6.3. Therefore, recall that we have m requestors, the number of slots in the schedule is at maximum 2^{m-1}. For each requestor, we store the period ($m-1$ bits) and the starting slot ($m-1$ bits). Since multiple requestors can be assigned the same slot, we store the order of the requestors in the execution ($\lceil \log_2 m \rceil$ bits). Finally, for the purpose of rate regulation, we store the maximum bundle limit $kmax_i$ for each requestor ($\log_2(\frac{2KB}{64B}) = 5$ bits for a request of $2KB$). Consequently, the data size overhead is small. In the worst-case, we need $m \times (2(m-1) + \lceil \log_2 m \rceil + 5)$ bits. As an example, a system with $m \leq 30$ requestors, the PMC requires less than 256 bytes to store the parameters.

4.6.3 Schedule Slots

As aforestated, the deployment framework utilizes the requirements prescribed by the requestors and the optimization objective of the system to produce a schedule that satisfies these requirements and optimizes for the selected objective. Figure 4.10 shows a schedule example for seven requestors, $R = \{r_1, r_2, ..., r_7\}$, with eight time slots. We use Figure 4.10 to illustrate the analysis provided in this section and Section 4.7. PMC assigns each requestor one or more slots within a schedule based on its LR_i and BWL_i requirements. For instance, r_1 is assigned slots: $slot_1$, $slot_3$, $slot_5$ and $slot_7$. This means that r_1 is granted permission to access the DRAM whenever its turn arises in these slots. It is worth noting that there is an order of requestors within a slot based on priorities assigned to requestors. In $slot_1$, the schedule grants permission to r_4 first, r_1 next, and r_3 last. When there are no requests from a particular requestor within a slot, PMC grants the next requestor the permission to send a request. The assignment of slots to requestors is harmonic ($s_i = 2^{q-1}$) where q is a positive integer. The rationale behind the harmonic-slot assignment is to schedule the requestors on a regular basis as it achieves 100% slot utilization. It also requires a smaller amount of memory to store the schedule in the controller as detailed in Section 4.6.2. The total number of slots in the schedule is n. This is a variable that is defined based on the system requirements, generally $n = 2^{m-1}$. In order to discover the smallest n, the framework selects a value of n. If it fails to generate a schedule satisfying the requirements, the

Table 4.3: Terms and brief descriptions.

Variable	Description
R	The set of requestors in the system.
r_i	Requestor number i in the system: $r_i \in R$.
$kmax_i$	Maximum number of bundles of a requestor r_i that are serviced per sub-request.
pr_i	r_i's relative priority.
LR_i	The memory access latency requirement of r_i.
BWL_i	The minimum bandwidth required by r_i.
s_i	Harmonic slots: total number of slots allocated to requestor r_i.
p_i	Harmonic period: the interval (in slots) between two successive executions of i. It is equal to the total number of slots divided by s_i.
x_{ik}	Indicator variable defined in equation 4.2, determines the total number of slots granted to requestor i.
y_{ij}	Indicator variable defined in equation 4.3, determines the slots granted to requestor i.
Y_j	The total number of requestors assigned to slot j.
w_j	The width of slot j in clock cycles.
W	The scheduling window: the total number of cycles of all slots. After W, the schedule is repeated.
UBD_i	The upper-bound latency incurred by a memory request from r_i.
LBB_i	The lower-bound bandwidth delivered to a requestor r_i.

framework increases n until we obtain a schedule that satisfies the requirements. To control the assignment of slots to requestors, we define two binary variables x_{iq} and y_{ij}. In Equation 4.2, $x_{iq} = 1$ only if r_i is assigned 2^{q-1} slots. Consequently, if we ensure that $\sum_{\forall q} x_{iq} = 1$, as we will see in Section 4.7.1, then we guarantee the harmonic property of slots. In Equation 4.3, y_{ij} identifies slots assigned to each requestor as it denotes whether r_i is assigned a particular slot j.

$$x_{iq} = \begin{cases} 1, & \text{if } s_i = 2^{q-1} \quad q \in \mathbb{Z}^+. \\ 0, & \text{otherwise.} \end{cases} \tag{4.2}$$

$$y_{ij} = \begin{cases} 1, & \text{if requestor } r_i \text{ is assigned to slot } j. \\ 0, & \text{otherwise.} \end{cases} \tag{4.3}$$

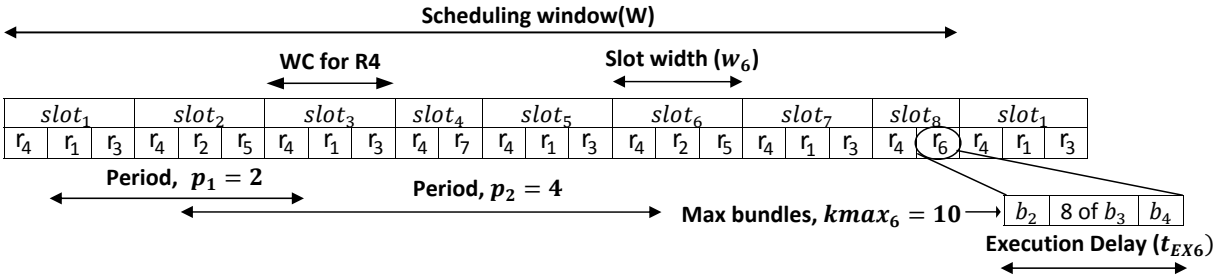


Figure 4.10: A schedule example.

Recall that the total number of requestors in the system is m. Using Equation 4.3, the total number of requestors that PMC grants access at slot j is calculated as $Y_j = \sum_{i=1}^{m} y_{ij}$, and the total number of slots s_i that PMC assigns to a requestor r_i is computed as $s_i = \sum_{j=1}^{n} y_{ij}$. Based on the slots assigned to requestors, each requestor has a harmonic period p_i. For example in Figure 4.10, requestor r_2 has $p_2 = 4$ slots.

4.7 Timing Analysis

We provide the timing analysis to upper-bound the latency incurred by any request to the DRAM, as well as lower-bound the delivered BW to any requestor. These bounds are necessary to achieve predictability. As aforestated, PMC decomposes a request into a number of sub-requests. Each sub-request is a sequence of consecutive bundle commands on the command bus that results in data transfers on the data bus. Figure 4.10 delineates that command sequence for a sub-request from r_6 in $slot_8$ which shows the *bundle time*. The bundle time of a sub-request represents the contribution of that sub-request to the interference latency of other sub-requests. Definition 4.2 formally defines this bundle time.

Definition 4.2. ***Bundle time.*** *The bundle time, t_{Bi}, of a sub-request from r_i is the time consumed by its bundles while it is performing the access to the DRAM. This time depends on the maximum number of consecutive bundles granted to r_i ($kmax_i$) and is calculated as:*

$$t_{Bi} = \begin{cases} t_{b_1}, & \text{if } kmax = 1 \\ t_{b_2} + (kmax_i - 2) \times t_{b_3} + t_{b_4}, & \text{if } kmax_i \geq 2 \end{cases}$$

Since PMC can assign multiple requestors to the same slot, the proposed schedule has a varying slot width. Equation 4.4 calculates the width of slot j, w_j. It is consists of the bundle times of the sub-requests granted access at slot j. Given that all slot widths are calculated using Equation 4.4, Equation 4.5 computes the total schedule window latency .

$$w_j = \sum_{i=1}^{m} (y_{ij} \times t_{Bi}) \tag{4.4}$$

$$W = \sum_{\forall j} w_j \tag{4.5}$$

Using Equation 4.4, Definition 4.3 formally defines the interference latency.

Definition 4.3. ***WC interference latency.*** *The worst-case interference latency suffered by a sub-request from r_i due to other sub-requests is defined as:*

$$t_{IFi} = p_i \times \text{MAX}_{\forall j}(w_j)$$

In the worst case, a requestor has to wait for p_i slots and each slot has the maximum width. p_i is the harmonic period of r_i as Table 4.3 defines.

Accounting for the execution latency (Definition 4.1) of the sub-request as well as the worst-case interference latency incurred due to other requests (Definition 4.3), Equation 4.6 upper-bounds the memory latency incurred by any sub-request accessing the DRAM.

$$UBL_{sub_i} = t_{IFi} + t_{EXi} \tag{4.6}$$

Recall that any request requiring a number of bundles $k_i > kmax$ is split into multiple sub-requests. Accordingly, the UBL for a request is computed as the UBL of its sub-requests multiplied by the number of sub-requests as shown in Equation 4.7. Equation 4.7 does not take the latency resulting from the interference of refresh commands into account. This is because Equation 4.7 calculates per request latency and it is not realistic to account for the refresh interference every request. However, it can be incorporated since the refresh operation is periodic and occurs every $tREFI$ cycles. A realistic approach to account for the refresh interference is to incorporate the refresh latency every designated number of requests. This can be done in a task-based analysis such as [28].

$$UBL_i = \left\lceil \frac{k_i}{kmax_i} \right\rceil \times UBL_{sub_i} \tag{4.7}$$

LBB_i is the lower-bound BW serviced to requestor r_i every W and is calculated by Equation 4.8, where $kmax_i \times BS$ represents the data size of the sub-request in bytes.

$$LBB_i = (kmax_i \times BS)/UBL_{sub_i} \tag{4.8}$$

4.7.1 Problem Formulation

We formulate the schedule generation problem as a mixed-integer non-linear optimization problem that can be solved using an appropriate optimization solver. We implement the optimization framework using Matlab [90]. As aforementioned, the framework enables the designer to build a schedule that meets the requirements of HRT and SRT requestors as well as optimizes for the system target simultaneously.

4.7.1.1 Target Function

The designer has the ability to optimize for one of four targets that we find practical for MCS: 1) the overall WCL, 2) the WCL incurred by some of the requestors (HRT requestors for example), 3) the overall BW provided by the DRAM, 4) the BW provided to some of the requestors (SRT requestors for example). As an example, the following formulation optimizes the schedule for the first target: minimizing the total WCL in the system.

$$\text{MIN}(\sum_{i=1}^{m} UBL_i)$$

4.7.1.2 Input Parameters

Recall from Figure 4.3a that the framework takes as an input the latency and bandwidth requirements of each requestor as well as requestor's relative priority if it exists.

$$LR_i, BWL_i, pr_i \qquad \forall i \textbf{ in } [1, ..., m]$$

4.7.1.3 Variable Parameters

For each requestor, r_i, the framework determines the optimal value of r_i's period, total number of slots assigned to r_i, the maximum number of bundles from r_i that PMC can grant an access to

DRAM consecutively. Finally, based on these values of all requestors, the framework determines the total number of slots in the schedule. These variable parameters construct the schedule that PMC executes and are respectively as follows.

$$p_i, s_i, y_{ij}, kmax_i, n \qquad \forall i \textbf{ in } [1, ..., m] \forall j \textbf{ in } [1, ..., n]$$

4.7.1.4 Constraints

The first constraint, C.1, ensures the harmonic property of the number of slots that PMC assigns to any requestor. The second constraint, C.2, asserts that the total number of assigned slots to any requestor is consistent with the selected harmonic number of slots chosen by the framework for that requestor. If the system has priorities between requestors, we provide the higher priority requestors with at least the same number of slots provided to the lower priority ones. Constraint C.3 accomplishes this target. However, the priority is an optional system parameter. Setting all priorities to 1, for example, makes the framework agnostic to this constraint. Priorities are also used to define the order of sub-requests within a slot. If no priorities are defined, the framework chooses an arbitrary order. The fourth and fifth constraints force the distributed-TDM characteristic in the schedule. They determine how to spread each requestor r_i over the slots to have a separation between each two successive executions to be exactly p_i. This is important to guarantee the UBL_i requirements. Constraint C.4 enforces that a request will get exactly one slot every p_i, and constraint C.5 asserts that the total number of assigned slots is equal to the harmonic number of slots determined by the framework. Constraints C.6 and C.7 assert that the LR and BWL requirements of all requestors are satisfied. These are optional parameters. If a requestor has no LR requirement, it can be set to infinity. If a requestor has no BW minimum requirements, it can be set to zero or one.

$$\forall i, l, k \textbf{ in } [1, ..., m] :$$

$$\sum_{\forall q} x_{iq} = 1 \tag{C.1}$$

$$\sum_{j=1}^{n} y_{ij} = s_i \tag{C.2}$$

$$pr_l < pr_k \implies p_l < p_k \tag{C.3}$$

$$\sum_{j=1}^{p_i} y_{ij} = 1 \tag{C.4}$$

$$\sum_{j=1}^{p_i} (y_{ij} \times \sum_{u=0}^{s_i-1} (y_{i,j+u\times p_i})) = s_i \tag{C.5}$$

$$UBL_i \leq LR_i \tag{C.6}$$

$$LBB_i \geq BWL_i \tag{C.7}$$

4.8 Experimental Evaluation

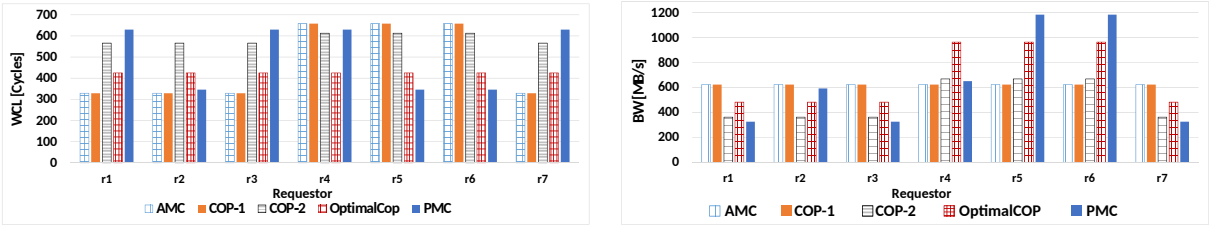
We extend MacSim, a multi-threaded architectural simulator [39] with the proposed PMC to manage accesses to a DDR3-1333 off-chip memory. We use a multi-core architecture model composed of x86 cores. The number of cores depend on the experiment. Each core has a private 16KB L1 and 256KB L2 caches, and a shared 1MB L3 cache. To compare the effectiveness of the proposed solution, we also implement two competitive MCs, the first one employs the conservative open-page policy [75] (COP), and the second one is AMC that employs the close-page policy [74]. In addition, we compare against a configurable system that combines the optimized TDM schedule in [83] and the COP. We use benchmarks from EEMBC-auto benchmark suite [91], which are representative for real-time applications. We conduct our evaluation by adopting two types of experiments: case-study system requirements and synthetic experiments.

4.8.1 Case-study: Multimedia System

System Configuration. We use a practical system with requirements modeled after the multimedia system in [83]. The system has seven requestors, r_1 to r_7, with different requirements. r_1 is an input device that writes the encoded media stream to the memory. r_2 and r_3 are the input and output cores/requestors respectively for a media engine decoder that decodes the media stream. r_4 and r_5 are the input and output cores/requestors respectively for a graphical processing unit (GPU). r_6 is an HDLCD-screen controller. Finally r_7 is the central processing unit (CPU) of the system. We first map these requirements to the DDR3 equivalent requirements and then adapt it for a single channel DDR, since it was originally proposed for a 4-channel memory system. Since the actual applications are not publicly available, we implement in-house workloads that match requirements of these tasks Table 4.4 also tabulates the requirements of each requestor. $LR = \infty$ means that the requestor has no LR requirements, while $BWL = 0$ models a requestor

Table 4.4: Multimedia processing system requirements.

Requestor	transaction size	LR_i (cycle)	BWL_i (MB/s)
r_1	128	∞	0
r_2	128	∞	384.9
r_3	128	∞	46.65
r_4	256	∞	500
r_5	256	816	250
r_6	256	816	250
r_7	128	∞	75



(a) WCL.

(b) Minimum BW.

Figure 4.11: Results for the multimedia processing system use-case.

with no BW requirements. The minimum transaction size in Table 4.4 is 128B; thus, we interleave across all banks of the DRAM. Accordingly, the bundle size (or the group size for case of COP) is 128B.

MC Configurations. To further validate the improvements we get in both the WCL and the average-case performance considering the proposed solution, we implement a memory controller that combines both the COP policy [75] and the optimized TDM schedule configuration in [83], which we call optimal COP. Optimal COP is able to assign different number of slots to the requestors based on the requirements. To compare the proposed PMC against optimal COP, we implement the proposed PMC as well as optimal COP in MacSim simulator. We assign a core for each requestor in Table 4.4.

Observations. Figures 4.11a and 4.11b show the experimental WCL and minimum BW, respectively for both PMC and the optimized COP. Results show that both MCs are able to meet the requirements. However, PMC shows better assignment of the resource based on the requirements.

4.8.2 Synthetic Experiments

To comprehensively evaluate PMC, we perform five sets of synthetic experiments.

1. We verify the capability of the proposed solution to satisfy different WCL and BW requirements. We carry this out by tuning the configurable parameters: the maximum number of consecutive bundles ($kmax_i$) and the schedule slots (s_i) of each requestor r_i.

2. We study the scalability of different MCs. We investigate the effect of the number of SRT requestors in the system on the WCL of HRT tasks.

3. We demonstrate the behaviour of PMC and competitive MCs with different transaction sizes.

4. We study the effectiveness of rank interleaving by comparing two versions of PMC: the first one supports only single-rank DRAMs while the second one interleaves across two ranks.

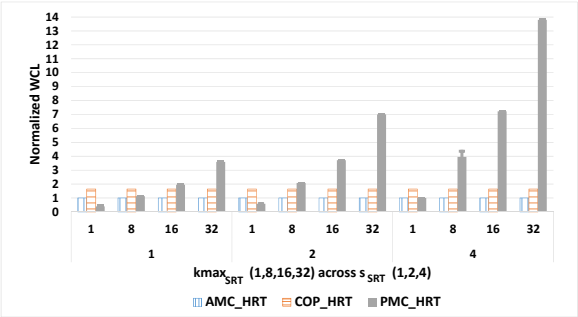
5. We compare the dynamic bank interleaving presented in Section 4.5.3 against static bank interleaving.

Recall that the role of the optimization framework is to determine the optimal values of the schedule parameters. Since in the synthetic experiments we are sweeping the configurable parameters (namely, $kmax$ and s), there is no need to use the optimization framework.

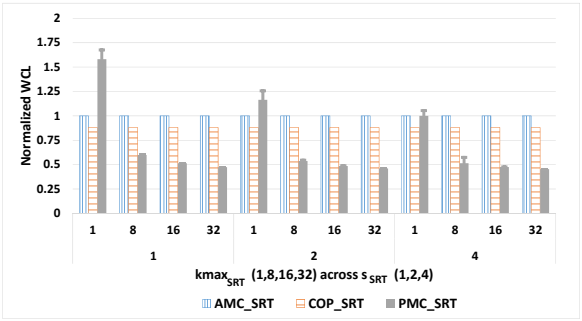
4.8.2.1 Varying PMC parameters

System Configuration. We deploy the following system configuration in the MacSim simulator. We use a multi-core architecture of five x86 cores (r_1 to r_5). r_1 is a HRT requestor with 64B memory transactions. r_2 to r_5 are SRT requestors with 2KB memory transactions. The used DRAM model is DDR-1333 [92]. Since the smallest transaction size is 64B, which can be obtained using four banks, we interleave across only four banks to avoid underutilizing the DRAM BW.

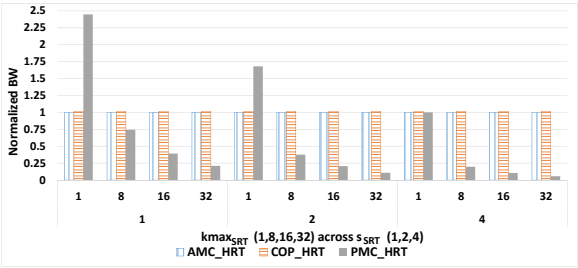
MC Configurations. AMC executes RR arbitration amongst the five requestors. COP executes a contiguous TDM schedule such that each requestor is assigned two consecutive slots. The 2-slot version of COP is chosen rather than the 1-slot version (where each requestor is granted only one slot) because it allows for locality exploitation among requests of the same core [75]. AMC and COP only support transactions up to the memory granularity. Hence, for both MCs, the 2KB transactions from SRT requestors are chopped into contiguous 64B transactions at the requestor side before sending them separately to the MC. On the other side, since PMC supports larger transaction sizes than the memory granularity, transactions are exposed to PMC as a



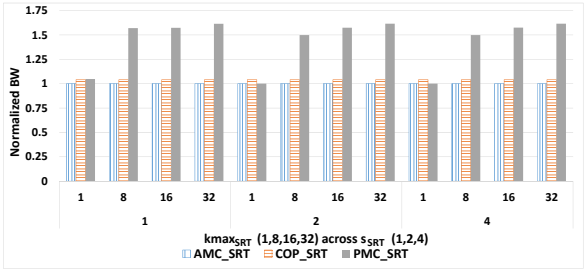
(a) WCL of the HRT requestor.



(b) WCL of one of the SRT requestors.



(c) BW of the HRT requestor.



(d) BW of one of the SRT requestors.

Figure 4.12: Effect of varying $kmax$ and s of SRT requestors.

whole without splitting them. PMC is capable of granting a different service to each one of the SRT requestor. However, for clarity, we simplify the experiment by granting all SRT requestors the same amount of service ($kmax$ and s). Since r_1 has 64B transactions, $kmax_1$ is set to 1. For SRT requestors (r_2 to r_5), we vary $kmax$ ($kmax_{SRT}$ in this context) to be 1, 8, 16 or 32. PMC's schedule consists of 4 slots. We grant the first sub-slot in each schedule slot for the HRT requestor (r_1), $s_1 = 4$. For SRT requestors, we vary s (s_{SRT} in this context) to be 1, 2 or 4.

Observations. Figures 4.12a and 4.12b depict the latency results of the HRT and one of the SRT requestors, respectively. We plot both the average- (solid coloured bars) and worst-case latencies (thinner T-sharp bars). Similarly, Figures 4.12c and 4.12d depict BW results. We normalize all results compared to the values obtained from AMC. Based on these results, we make the following observations:

(1) Both AMC and COP have a fixed WCL since they have a fixed schedule and a bounded transaction size. In contrast, PMC has the capability of achieving different WCL and BW for different use-cases or requirements. As Figures 4.12a–4.12d illustrate, this is attained by tuning the configurable parameters.

(2) Results highlight the main novelty of PMC: *exploring the trade-off between temporal and BW requirements of different tasks to provide the optimal MC behaviour*. Assigning a higher $kmax$ for SRT requestors improves their average-latency (Figure 4.12b) and BW (Figure 4.12d). However, it increases the WCL of HRT requestors (Figure 4.12a). Contrarily, a lower $kmax_{SRT}$ will reduce the WCL of HRT requestors by throttling the BW serviced to SRT requestors. Similar effect occurs by changing the number of granted slots to each SRT requestor s_{SRT}. The optimal ($kmax$ and s) pair per requestor depends on the use-case requirements and is determined by the provided optimization framework.

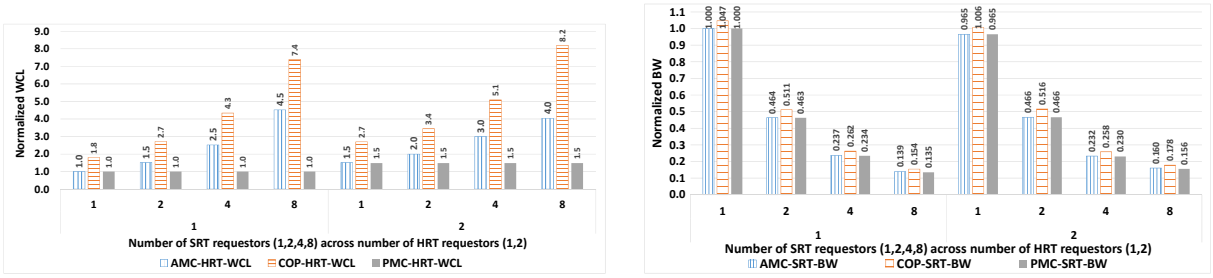
(3) Any system requirements that can be satisfied using AMC or COP are satisfied by the proposed mixed-policy PMC. This is because PMC encompasses both behaviours of AMC and COP. Setting $kmax = 1$ for all requestors and assigning SRT requestors the same number of slots as HRT ones ($s_{SRT} = 4$) results in a behaviour similar to AMC. Correspondingly, setting $kmax = 2$ and assigning SRT requestors the same number of slots as HRT ones achieves a behaviour similar to COP.

(4) Figures 4.12a and 4.12b delineate the memory latency bounds for PMC (thinner T-sharp bars) obtained from the static analysis. Results show that the calculated bounds are safe since all obtained WCL measurements are less than their corresponding bounds.

4.8.2.2 WCL scaling: Effect of Number of Requestors

System Configuration. Recall that the highest priority target of MCS is to meet the temporal requirements of the most critical requestors (HRT). In these experiments, we investigate the ability of MCS to satisfy the WCL requirements of HRT, while increasing the number of SRT requestors in the system. Similar to the previous system configuration, the HRT requestors issue 64B transactions and the SRT requestors issue 2KB transactions. We conduct two sets of experiments. The first set has a single HRT requestor, while the second one has 2 HRT requestors. In both sets, the number of SRT requestors vary from 1 to 8.

Observations. Figure 4.13a represents the WCL of the HRT requestor(s), while Figure 4.13b represents the BW of the SRT requestors. We normalize all results compared to the AMC results for one HRT requestor and one SRT requestor. (1) Figure 4.13a demonstrates that PMC successfully achieves the target of the experiment by providing a fixed WCL for the HRT requestor(s) regardless of the number of SRT requestors. This is by virtue of the configuration capability of both the rate regulator ($kmax$) and the arbitration schedule (s). We configure $kmax$ and s for all requestors such that each HRT requestor is assigned a sub-slot in all schedule slots, while only one SRT requestor is assigned a sub-slot in a schedule slot. In addition, we set $kmax = 1$ for all SRT requestors. In consequence, the WCL of HRT requestors remains the same regardless of the number of SRT requestors in the system. In contrast, for the one HRT requestor experiment,



(a) WCL of a HRT requestor.

(b) BW of a SRT requestor.

Figure 4.13: Effect of varying number of requestors.

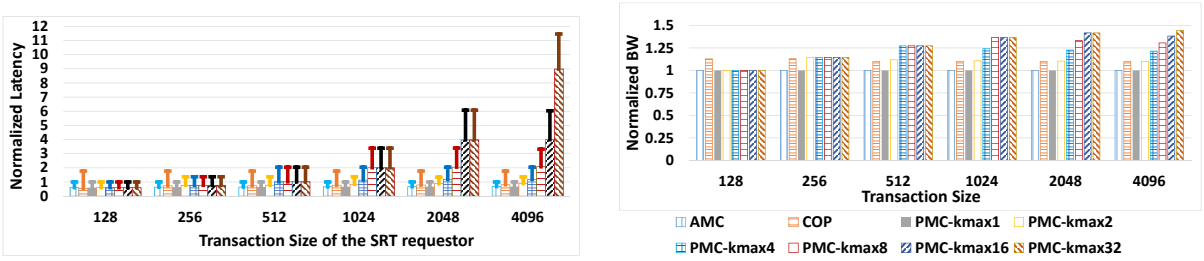
the WCL of the HRT requestor increases by up to 352% and 310% in comparison to a single SRT requestor case in AMC and COP, respectively. Similarly, for the two HRT requestors experiment, the WCL of each HRT requestor increases by up to 166% and 204% in AMC and COP, respectively.

(2) We observe that for a system with more than one SRT requestor, the BW delivered to the SRT requestors by PMC is less than that delivered by AMC or COP MCs. This is because we set the values of $kmax$ and s to minimize the WCL of HRT requestors. Hence, we sacrifice part of the service delivered to SRT requestors. If the BW delivered by PMC to SRT requestors is not satisfying their requirements, another configuration should be selected to relax the constraint of having a fixed WCL of the HRT requestor, and increase the BW delivered to SRT requestors. Again, this emphasizes the potential of the proposed framework to have different schedules for different system requirements.

(3) Finally, we observe that COP offers higher bandwidth for SRT requestors at the expense of higher WCL of HRT requestors compared to AMC and PMC. This is because COP assigns two consecutive slots to each requestor. SRT requestors usually utilize these slots and send requests that exploit row locality as they are memory intensive due to the large-size requests (each 2KB request is split into 32 successive 64B accesses). Therefore, the BW of SRT requestor increases. On the other side, HRT requestors, in the worst case, have to wait for two slots per SRT requestor which increases their WCL.

4.8.2.3 Effect of the Transaction Size

To study the effect of various MCs and system parameters, we conducted all previous experiments assuming a single transaction size for both SRT (2KB) and HRT (64B) requestors. In reality, while the transaction size of a core request is usually determined by the line size of its



(a) Memory Latency.

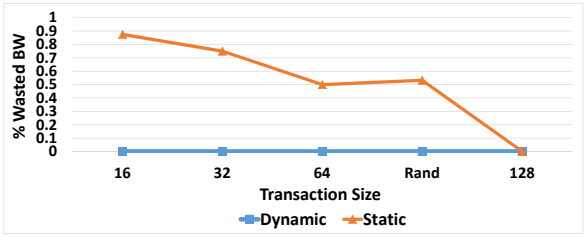
(b) BW.

Figure 4.14: Effect of the transaction size.

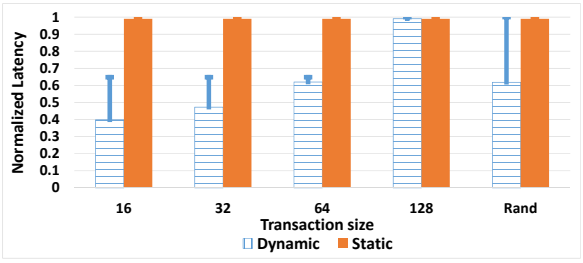
last-level cache, other requestor types such as DMAs and IO processing elements can issue requests with different transaction sizes. Hence, in this experiment we study the effect of different transaction sizes on the behaviour of experimented MCs.

System Configuration. We experiment using one HRT requestor and one SRT requestor. The HRT requestor issues a transaction size of 128B in all experiments in this set, while the SRT requestor issue a different transaction size in each experiment, which varies between 128B and 4KB.

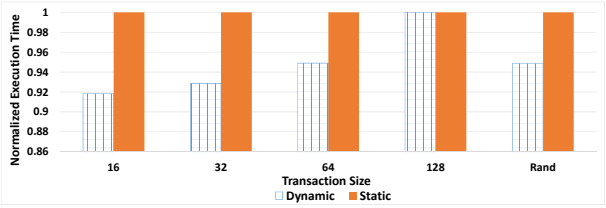
MCs Configuration. Since the minimum transaction size in this set of experiments is 128B, we interleave across the 8 banks for all MCs. In addition, we experiment using different $kmax$ configurations. We plot both the average- (solid coloured bars) and worst-case (thinner T-sharp bars) latencies for the HRT requestor in Figure 4.14a and the BW delivered to the SRT requestor in Figure 4.14b. The legend PMC-kmax$i$ represents a PMC configuration with $kmax = i$. We normalize all values based on the experimental WCL of the AMC controller. ***Observations.*** (1) Figure 4.14 confirms the aforementioned observation that PMC encompasses the behaviour of both AMC and COP by comparing the behaviour of PMC-kmax1 against AMC, and PMC-kmax2 against COP. (2) COP with 2 consecutive slots assigned to each requestor has higher WCL than AMC, while it utilizes these two slots to increase average-case BW by keeping the row open as much as possible. (3) The configurability of PMC provides the ability to provide different WCLs and BWs by changing the $kmax$. The framework chooses the suitable $kmax$ to satisfy requirements of all tasks. (4) We deduce from Figure 4.14 that there is no effect from assigning a $kmax$ value to a requestor higher than the sufficient value to serve all its required data size in one access. For example, for a transaction size of 512B, assigning $kmax > 4$ to the SRT requestor has no effect on WCL nor BW compared to $kmax = 4$. Recall that we interleave across all 8-banks, and the bundle size is 128B. Accordingly, assigning $kmax = 4$ to the SRT requestor, it is able to issue four consecutive bundles to transfer a data of 512B.



(a) Wasted BW.



(b) Memory latency.



(c) Total execution time.

Figure 4.15: Dynamic *vs.* static bank interleaving.

4.8.2.4 Supporting Dynamic Bank Interleaving

We add the dynamic bank interleaving support to PMC. Based on the transaction size, PMC decides the number of banks to interleave across and hence, selects the appropriate bundles to access the DRAM.

Experiment Setup. We run different experiments in which we vary the number of requestors and the issued transaction sizes. For clarity, we show the results of experiments with one requestor that issues a different transaction size in each experiment (16B, 32B, 64B, *Rand* and 128B). In the experiment with *Rand* transaction size, the requestor issues requests with a random transaction size that is amongst the following set: $\{16\text{B}, 32\text{B}, 64\text{B}, 128\text{B}\}$.

MC Configurations. We perform the experiments on both the PMC version with static bank interleaving, where PMC interleaves across the 8 available banks and the PMC version with dynamic bank interleaving. We delineate the results in Figures 4.15a–4.15c.

Observations. (1) Figure 4.15a depicts the percentage of non-utilized BW as defined by Equation 4.1 in Section 4.5.3. *Static bank interleaving* transfers a fixed data size each transaction (128B in our experiments). Therefore, the lower the actual requested data size by the transactions is, the higher the non-utilized BW is. As Figure 4.15a shows, the non-utilized BW reaches up to 87% for a requestor with 16B transactions. In contrast, *dynamic bank interleaving* adjusts

the number of banks to interleave across in each transaction to transfer the requested data size. Hence, as Figure 4.15a illustrates, there is no non-utilized BW.

(2) Figure 4.15b presents both the average-case latencies (solid coloured bars) and the experimental worst-case latencies (thinner T-sharp bars). We normalize all values based on the experimental WCL of the *static interleaving* configuration. From Figure 4.15b, we make the following observations. (2.a) For a requestor with a fixed transaction size that is less than the memory granularity (16B, 32B and 64B in Figure 4.15b), dynamic bank interleaving achieves both better worst- (35% less) and average-case (38% to 60% less) latency compared to static interleaving across all available banks. (2.b) For a requestor that generates random transaction sizes ($Rand$ in Figure 4.15b), the worst-case latency is the same as the static interleaving. This WCL is suffered by the 128B transactions as they require interleaving across all available banks. However, Figure 4.15b highlights that even for a requestor with random transaction sizes *dynamic interleaving* outperforms *static interleaving* on average-case by 40%.

(3)Figure 4.15c depicts the execution time of the application running on the requestor. We normalize all values compared to the execution time of the *static interleaving* configuration. As Figure 4.15c illustrates, *dynamic interleaving* decreases the total execution time of the application by 5% to 8%.

4.8.2.5 Supporting Rank Interleaving

We test the effectiveness of rank interleaving in decreasing both average- and worst-case latency compared to single-rank PMC.

System Setup. In order to quantify the effect of rank interleaving on eliminating switching latency, we perform this comparison using a single core to eliminate latencies due to interference from other requestors. As discussed in Section 4.5.4, the rank interleaving mechanism is effective only for requests accessing 8 banks. Hence, in this experiment, we interleave across 8 banks for the single-rank case. We sweep the transaction sizes from 128B to 4KB.

MC Configurations. We compare the PMC with single-rank bundles (*PMC-1RNK*) against the multi-rank bundles (*PMC-2RNK*). Figure 4.16 illustrates the results of this comparison. It depicts both the average-case latencies (solid coloured bars) and the experimental worst-case latencies (thinner T-sharp bars). We normalize all values based on the experimental WCL of *PMC-1RNK*.

Observations. Based on Figure 4.16, we highlight the following observations. (1) For small size requests (128B and 256B), interleaving bundles across different ranks results in better worst-case latency compared to mapping the bundles to a single rank. Given that HRT requestors usually issue small size requests (cache line size), rank interleaving is crucial to decrease their

WCL. Moreover, for the $128B$ case, the average-case latency of *PMC-2RNK* is also less than that of *PMC-1RNK*. This is because requests accessing different ranks do not suffer from bus switching time.

(2) There exists a considerable difference between the average-case and worst-case latencies of single requestor accesses the DRAM for *PMC-1RNK* (up to 10% difference for 128B transaction size) while they coincide for *PMC-2RNK*. This is because, in worst case, *PMC-1RNK* assumes a switching latency between every two successive requests, while in average-case the switching latency may be less. On the other hand, *PMC-2RNK* does not encounter a switching latency. Moreover, the rank-to-rank switching latency is incorporated in the bundle execution time since each bundle is accessing two ranks. Recalling that in this experiment only a single requestor accessing the DRAM, the only source of unpredictability is the data bus switching time. Hence, both average- and worst-case latencies are expected to be the same.

(3) The latency gaps between average- and worst-case latencies in *PMC-1RNK* diminishes by increasing the transaction size (from 10% for 128B transaction size to 0.4% for 4KB transaction size). We interpret this observation by analysing the ratio between the execution and switching latencies. By increasing the transaction size, the execution latency increases, while the switching latency remains the same. As a result, the impact of the switching latency on the total memory latency diminishes. (4) Increasing the transaction size, *PMC-2RNK* incurs larger WCL than *PMC-1RNK*. As aforestated in Section 4.5.4, this is expected for the following reason. The number of bundle 3 increases by the increase in the transaction size. Recall that tb_3 for *PMC-2RNK* is larger than tb_3 for *PMC-2RNK* because of the $tRTRS$ constraint. Accordingly, increasing number of bundle 3 increases the worst-case latency of *PMC-2RNK* compared to *PMC-1RNK*.

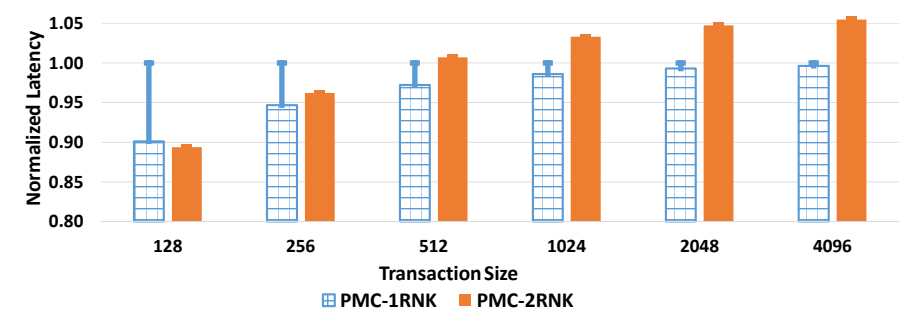


Figure 4.16: Effect of rank interleaving on memory latency.

4.9 Summary

In this chapter, we presented PMC, a programmable DRAM MC for MCS, and an optimization framework to provide optimal schedules for different sets of applications running on these systems. PMC supports an arbitrary number of criticality levels by enabling the MCS designer to specify memory requirements per task. In addition, the framework optimizes the schedule for different MCS memory targets such as total worst-case latency or bandwidth. We also promoted a novel implementation of TDM schedule that enables lower worst-case latencies than contiguous TDM, while it has a lower area overhead than distributed TDM. PMC allows different requestors to issue memory requests with different transaction sizes. This is important for practical systems such as media processing systems, especially with multi-core architectures. We implemented a mixed-page policy scheme that dynamically switches between close- and open-page policies. By exploiting locality, the proposed policy reduces the worst-case latency of requests while increasing the average-case performance compared to state-of-the-art MCs. Finally, we presented a complete static analysis to provide upper bounds on the latency, and lower bounds on the BW serviced to any requestor.

Chapter 5

Reverse Engineering Embedded DRAM Memory Controllers through Latency-based Analysis

We explore techniques to reverse-engineer properties of DRAM MCs. This includes page policies, address mapping schemes and command arbitration schemes. There are several benefits to knowing this information: they allow analysis techniques to effectively compute worst-case bounds, and they allow for platform-aware optimizations at the operating system, source-code, and compiler levels. We develop a latency-based analysis, and use this analysis to devise algorithms and C programs to extract properties of MCs. We show the effectiveness of the proposed approach by reverse-engineering the implementation details of the MC deployed in the XUPV5-LX110T platform from Xilinx. Furthermore, in order to cover a breadth of page policies, address mappings and command arbitration schemes, we explore our technique using a micro-architecture simulation framework and document our findings.

5.1 Introduction

Modern computing systems implement a memory hierarchy with a combination of on-chip scratchpads, caches, and off-chip DRAMs [93]. This hierarchy is a critical component of all computing systems, employed in server, embedded, desktop, and mobile systems [94]. Realizing sufficient information about the implementation details of this hierarchy has several implications on various areas of research. Leveraging this information would assist architecture simulators to be more accurate [95], and assist compilers to provide platform-aware optimizations [96,97]. In addition,

it enables operating systems to allocate memory in a certain way to provide high performance or task isolation [98]. From a security perspective, exposure to this information enables the researchers to identify existing vulnerabilities in the memory system, which can lead to covert- and side-channel attacks [99, 100]. Moreover, it facilitates efficient construction of hardware disturbance errors such as the well-known DRAM rowhammer attack [101]. For embedded systems, information about the memory hierarchy is necessary to allow WCET analysis techniques to account for latencies incurred during memory accesses [28, 95]. Unfortunately, manufacturers consider the implementation details of the memory hierarchy as intellectual property; hence, this information is not publicly available neither for caches [95] nor for DRAMs [98].

There are several research efforts that reverse-engineer cache properties [95, 102–107], but, there is limited work that does the same for main memories [98, 108, 109]. Therefore, there is a need to devise approaches to expose implementation details of the main memory system; thereby, it is the focus of this chapter. The main memory system composes of one or more DRAM channels and a MC managing accesses to the DRAM. A MC comprises three main components: address mapping, page policy, and arbitration scheme. Prior efforts [98, 108, 109] partially reverse-engineer properties of the MC. In particular, they discover the address mapping schemes. However, they do not discover page policies or command arbitration schemes that can provide further opportunities for research in all of the aforementioned aspects. For instance, authors in [28] assume that all properties of the MC including the page policy and arbitration are known a priori. They use this information to provide bounds on memory interferences in multi-core systems. This provides evidence of the advantages of knowing properties of the MCs. However, the techniques to reverse-engineer important properties of MCs remains an unexplored challenge.

5.1.1 Contributions

In response to the highlighted challenge, we develop a latency-based analysis to reverse-engineer essential properties of the MC. We discover commonly used page policies, address mapping schemes, and command arbitration schemes. Our technique relies on deriving best- and worst-case latency equations for memory accesses to the MC (Section 5.3). We use this analysis to develop algorithms for micro-benchmarks that can elicit properties of the MC (Section 5.4). We show the effectiveness of the proposed approach by reverse-engineering the implementation details of the MC deployed in the XUPV5-LX110T platform from Xilinx. Moreover, since most hardware platforms typically have a fixed set of MC policies, we deliberately experiment with a micro-architectural simulation framework MacSim [39] interfaced with a comprehensive DRAM simulator called DRAMSim2 [110] to enable a thorough exploration of MC configurations. Finally, we highlight the potential exploitations that the reverse-engineering of MC properties in-

spires (Section 5.5).

5.2 Related Work

Researchers have proposed various techniques to infer properties of caches using measurement-based analysis [95, 102–107]. We broadly classify these approaches into those that use performance counters present in the platforms [95, 102–104], and those that rely on latency analysis [105–107]. The approaches proposed in [95, 102–104] make use of performance counters available in current platforms to infer properties of the cache hierarchy. While [102, 104] identify least recently used (LRU) replacement policies and variants of LRU such as pseudo-LRU (PLRU) and fill PLRU, a recent work by Abel and Reineke [95] uses block order maintained in the cache sets due to cache hits and misses to distinguish between LRU, first-in first-out (FIFO), and random replacement policies. Latency analysis approaches [105–107] measure access latencies to the memories to discover their properties. This approach is necessary when performance counters are either unavailable or do not provide sufficient statistics for inference. For example, authors in [105] infer cache properties of an NVIDIA GT200 GPU via latency analysis because performance counters were unavailable.

Recent works such as [98, 108, 109] infer certain properties of the MC in an effort to propose novel virtual-to-physical page allocations. Yun et al. [98] propose a new virtual-to-physical memory allocation scheme by first inferring the mapping between virtual address bits and physical bank bits for the Intel Xeon processor using latency-based analysis. Park et al. [108] employ similar latency based analysis to identify channel, rank, and bank bit mapping between virtual and physical addresses. However, we find the approach followed by [108] is suitable for mappings where all the bits assigned to a certain group (such as bank, ranks or channels) are contiguous. This approach will not be able to reveal details of distributed address mapping schemes. Recently, authors in [109] reverse engineered the address mapping details of various Intel architectures. They use both physical probing and latency-based analysis. However, all these approaches [98, 108, 109] do not infer other important properties of the MC such as the page policy, and command arbitration schemes that are essential in understanding the temporal behaviour of the MC.

In Chapter 5, we attempt to do this for the most common page policies, address mappings, and command arbitration schemes.

5.3 Memory Latency Analysis

When the MC grants the logical memory requests (Definition 5.1) access to the DRAM, it converts the logical memory requests into physical memory requests. A physical memory request (Definition 5.3) consists of two components: the physical address (Definition 5.2), and a sequence of low-level DRAM commands. The address mapping policy translates the logical memory address to the physical memory address.

Definition 5.1. ***Logical memory request.*** *A logical memory request is a 2-tuple* $lr = \langle la, o \rangle$ *where* la *is a* LW *bits wide logical memory address* $la \in \{0,1\}^{LW}$ *and* $o \in \{R, W\}$ *designates a read or write access operation.*

Definition 5.2. ***Physical address.*** *A physical address* $pa = \langle cn, rnk, bnk, rw, cl \rangle$ *is* PW *bits wide. It is composed of* CNW *channel bits,* RKW *rank bits,* BKW *bank bits,* RWW *row bits and* CLW *column bits, respectively.*

Definition 5.3. ***Physical memory request.*** *A physical memory request is a 2-tuple* $pr = \langle pa, cs \rangle$ *such that* pa *is the physical address and* cs *is a sequence of DRAM commands.*

Definition 5.4. ***Arrival time.*** *The arrival time* t_i *is the time-stamp at which the first DRAM command of* pr_i *arrives at the command queue.*

Definition 5.5. ***Finish time.*** *The finish time* f_i *of a physical request* pr_i *is the time-stamp at which* pr_i *starts its data transfer.*

Definition 5.6. ***Access latency.*** *The access latency of the* i^{th} *physical request* pr_i *is defined as* $l_i = f_i - t_i$.

The commands issued by the MC to the DRAM adhere to certain timing constraints based on a DRAM access protocol. These timing constraints affect the access latency of any request to the DRAM. If the arrival time of pr_i is such that pr_i will not incur any waiting latency due to timing constraints between commands of pr_i and commands of previous requests, then pr_i will incur the best-case access latency. Figure 5.1 illustrates two physical requests, pr_1 and pr_2 with their arrival times t_1 and t_2, latencies l_1 and l_2 and finish times f_1 and f_2. Let the MC be initially idle and pr_1 arrives at time-stamp 0 (t_1=0). Hence, in Figure 5.1, pr_1 trivially satisfies the timing constraints and the MC issues A1 immediately. However, t_2 does not satisfy the timing constraints as pr_2 arrives before the P1 is issued; therefore, the MC must delay issuing A2 to satisfy the timing constraints.

The arrival time t_i depends on several factors that the MC cannot control. For example, delays incurred due to pipeline stalls or the interconnect. As a result, we study the effect of

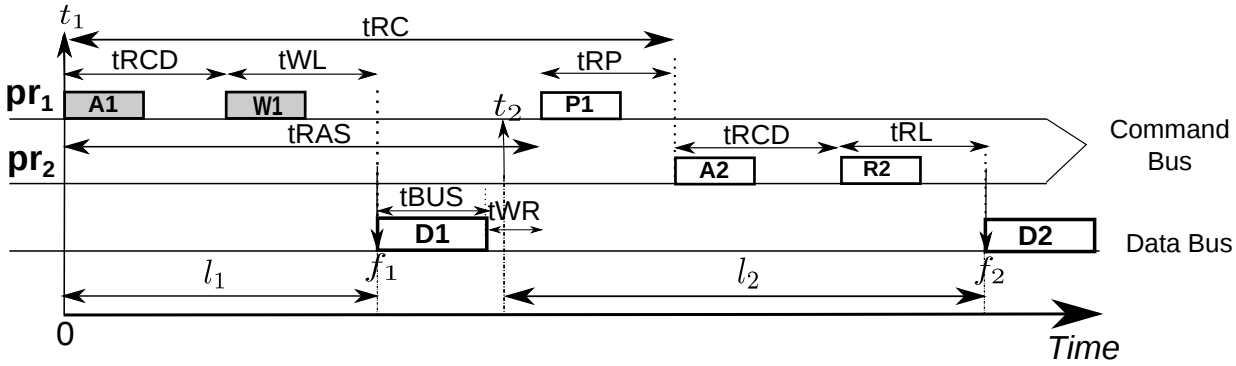


Figure 5.1: A write access followed by a write or read access targeting the same bank and rank. A: ACTIVATE, W: WRITE, R: READ, P: PRE, D: DATA.

arrival times on access latencies experienced by physical requests. Let pr_1 = $\langle pa_1, cs_1 \rangle$ and pr_2 = $\langle pa_2, cs_2 \rangle$ be two successive physical requests. Our approach presents an analysis to derive the access latency for pr_2, its best- (l_2^{BEST}), and worst-case access (l_2^{WORST}) latency bounds. The analysis is done under the assumption that the DRAM MC is initially in an idle state; hence, there are no active rows in the row buffers. Recall that we use R command for a read CAS and W command for a write CAS. If the access latency analysis is agnostic to the request type, then R and W have the same effect on the latency. Therefore, we denote the access command simply as CAS, and the timing constraint between the CAS and the start of the data transfer as tCL. Since $tBUS$ constraint includes the $tCCD$ constraint in all DDR modules, throughout this chapter we let $tBUS \geq tCCD$.

5.3.1 Proof Strategy

We highlight the strategy we follow to obtain the best- and worst-case access latencies. We then introduce an example to apply this strategy for two accesses with same access type to two different banks in the same rank.

Theorem 5.1. *The effect of DRAM timing constraints on a request latency can be expressed generally as a function of two conditions on the arrival time of that request,* $cond_1$ *and* $cond_2$*, such that the best-case latency for* pr_2 *occurs when* $t_2 \geq \hat{t_2}$*, where* $\hat{t_2} = \text{MAX}(cond_1, cond_2)$.

Proof. Let pr_1 and pr_2 be successive requests to a MC in the idle state, and pr_1 arrives at 0 ($t_1 = 0$). Hence, the first command of pr_1 (A1) can be issued immediately. However, pr_2 has to satisfy the timing constraints between commands of pr_1 and pr_2 before it can issue its first

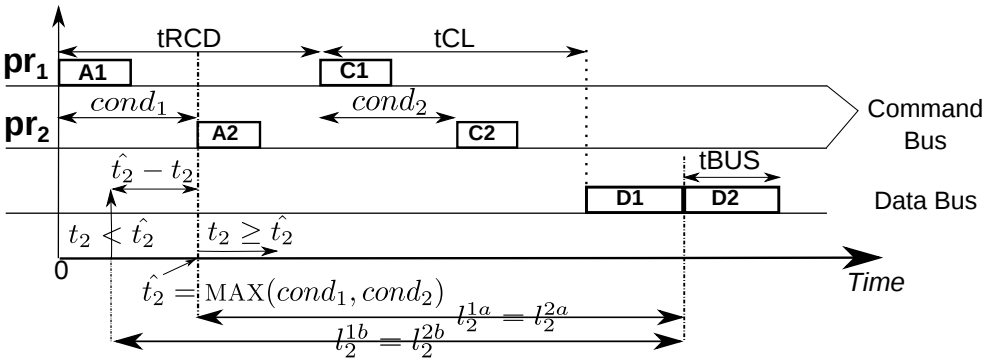


Figure 5.2: The two conditions controlling the issuance of the first command of pr_2. C1 and C2 represent CAS1 and CAS2 commands respectively.

command. The observation we make in this proof strategy is that these timing constraints can be combined into two conditions. These two conditions, denoted as $cond_1$ and $cond_2$, must be satisfied before the first command of pr_2 can be issued. Figure 5.2 depicts an example of these two conditions. $cond_1$ in Figure 5.2 represents the timing constraints between A1 and A2 commands, while $cond_2$ represents the constraints between CAS1 and CAS2 commands. Suppose that $cond_1 \geq cond_2$, then $\hat{t_2} = cond_1$. There are two cases based on the arrival time of pr_2.

Case 1a: When $t_2 \geq \hat{t_2}$, then $cond_1$ is satisfied. Since $cond_1 \geq cond_2$, $cond_2$ is also satisfied. Therefore, commands of pr_2 will not incur any latency due to commands of pr_1. Let the latency of pr_2 in this case be l_2^{1a}.

Case 2a: When $t_2 < \hat{t_2}$, then $cond_1$ is not satisfied. Hence, the MC delays the issuance of the first command of pr_2 by $\hat{t_2} - t_2$ resulting in an access latency of $l_2^{2a} = (\hat{t_2} - t_2) + l_2^{1a}$. Now, Suppose that $cond_1 < cond_2$, then $\hat{t_2} = cond_2$. There are again two cases based on the arrival time of pr_2.

Case 1b: When $t_2 \geq \hat{t_2}$, then $cond_2$ is satisfied. Since $cond_2 > cond_1$, $cond_1$ is also satisfied. Therefore, commands of pr_2 will not incur any latency due to commands of pr_1. Let the latency of pr_2 in this case be l_2^{1b}. Note that $l_2^{1b} = l_2^{1a}$.

Case 2b: When $t_2 < \hat{t_2}$, then $cond_2$ is not satisfied. Hence, the MC delays the issuance of the first command of pr_2 by $\hat{t_2} - t_2$ resulting in an access latency of $l_2^{2b} = (\hat{t_2} - t_2) + l_2^{1b}$. Note that $l_2^{2b} = l_2^{2a}$. Since $l_2^{1a} < l_2^{2a}$ and $l_2^{1b} < l_2^{2b}$, the arrival time for pr_2 producing the best-case latency occurs when $t_2 \geq \hat{t_2}$ with $\hat{t_2} = \text{MAX}(cond_1, cond_2)$. □

The following corollary uses results of Theorem 5.1 to compute the access latency l_2.

Corollary 5.1. *The latency of pr_2 at any given arrival time t_2 when $\hat{t_2} = \text{MAX}(cond_1, cond_2)$ is*

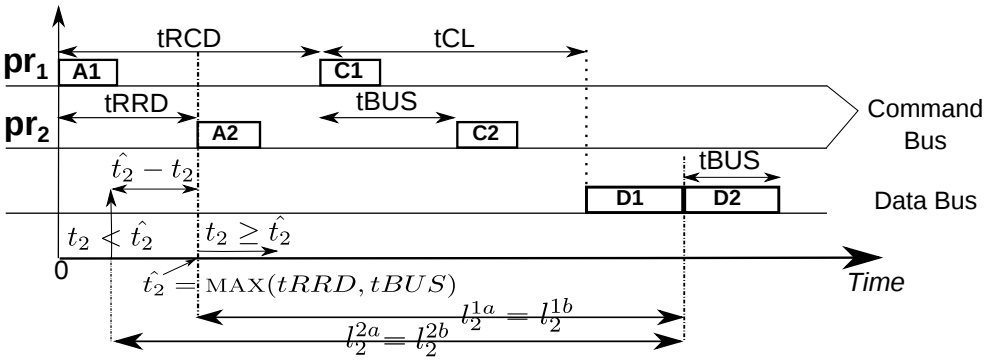


Figure 5.3: Two accesses with same access type to two different banks in the same rank.

given by:

$$l_2 = \text{MAX}(\hat{t_2} - t_2, 0) + l_2^{1a}.$$

Substituting $t_2 \geq \hat{t_2}$ in Corollary 5.1 will give the best-case latency $l_2^{BEST} = l_2^{1a}$, while substituting $t_2 = 0$ will give the worst-case latency $l_2^{WORST} = \hat{t_2} + l_2^{1a}$.

5.3.2 Example 1: Two accesses with same access type to two different banks in the same rank

For two requests with the same access type, $cs_1 =$ CAS1 and $cs_2 =$ CAS2 such that CAS1 and CAS2 are of the same type (both should be either R or W), Figure 5.3 shows the timing diagram for this sequence.

Theorem 5.2. *The best-case latency for pr_2 occurs when $t_2 \geq \hat{t_2}$, where $\hat{t_2} = \text{MAX}(tRRD, tBUS)$.*

Proof. This proof is obtained by substituting $cond_1 = tRRD$ and $cond_2 = tBUS$ in the proof strategy in subsection 5.3.1. Given that the MC is initially idle, and pr_1 arrives at 0 ($t_1 = 0$), the DDR specifications state that $tRRD, tRCD, tCL$ and $tBUS$ in Table 4.1 (Section 4.2) should be satisfied before issuing A2 and CAS2. These constraints are shown in Figure 5.3. Suppose that $tRRD \geq tBUS$, then $\hat{t_2} = tRRD$. There are two cases based on the arrival time of pr_2.
Case 1a: When $t_2 \geq \hat{t_2}$, A2 command can be issued immediately and after $tRCD$ cycles the MC issues CAS2. Then, $l_2^{1a} = tRCD + tCL$, where tCL cycles are necessary before the starting of data transfer.
Case 2a: When $t_2 < \hat{t_2}$, $l_2^{2a} = (\hat{t_2} - t_2) + tRCD + tCL$.
Now, suppose that $tBUS > tRRD$ such that $\hat{t_2} = tBUS$. There are again two cases based on the arrival time of pr_2.

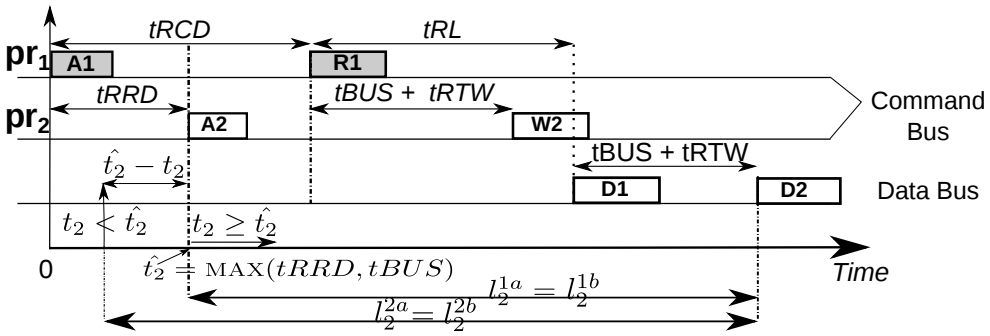


Figure 5.4: A read access followed by a write access targeting different banks in the same rank.

Case 1b: When $t_2 \geq \hat{t_2}$, $l_2^{1b} = tRCD + tCL$.
Case 2b: When $t_2 < \hat{t_2}$, $l_2^{2b} = (\hat{t_2} - t_2) + tRCD + tCL$.

Since $l_2^{1a} < l_2^{2a}$ and $l_2^{1b} < l_2^{2b}$, the arrival time for pr_2 producing the best-case latency occurs when $t_2 \geq \hat{t_2}$ with $\hat{t_2} = \text{MAX}(tRRD, tBUS)$. □

Corollary 5.2. *The latency of pr_2 at any given arrival time t_2 when $\hat{t_2} = \text{MAX}(tRRD, tBUS)$ is given by:*

$$l_2 = \text{MAX}(\hat{t_2} - t_2, 0) + tRCD + tCL.$$

When $t_2 \geq \hat{t_2}$ in Corollary 5.2 will give the best-case latency $l_2^{BEST} = tRCD + tCL$, while substituting $t_2 = 0$ will give the worst-case latency $l_2^{WORST} = \hat{t_2} + tRCD + tCL$.

5.3.3 Example 2: Two accesses with different access type to two different banks in the same rank

Figure 5.4 shows a read request $pr1$ followed by a write request pr_2 such that $cs_1 = [\mathsf{A1}, \mathsf{R1}]$ and $cs_2 = [\mathsf{A2}, \mathsf{W2}]$.

Theorem 5.3. *The best-case latency for pr_2 occurs when $t_2 \geq \hat{t}$, where $\hat{t} = \text{MAX}(tRRD, tBUS + tRTW)$.*

Proof. The MC has to reverse the data bus direction from read to write. This adds additional $tRTW$ cycles between R1 and W2 as shown in Figure 5.4. Suppose that $tRRD \geq (tBUS + tRTW)$, then $\hat{t} = tRRD$. There are two cases based on the arrival time of pr_2.
Case 1a: When $t_2 \geq \hat{t}$, the $tRRD$ timing constraint is trivially satisfied and A2 can be issued immediately. Since $t_2 \geq tRRD \geq tBUS + tRTW$, $t_2 + tRCD$ satisfies the $tBUS + tRTW$

Table 5.1: Best and worst-case latencies.

Latency Equation	Configuration	Reference $\hat{t_2}$
$l_2^{WORST} = \hat{t_2} + tRCD + tCL$	Different Ranks	$tBUS + tRTRS$
	Different Banks and RR/WW	$\text{MAX}(tRRD, tBUS)$
$l_2^{BEST} = tRCD + tCL$	Different Banks and RW	$\text{MAX}(tRRD, tBUS + tRTW)$
	Different Banks WR	$\text{MAX}(tRRD, tWL + tBUS + tWTR)$
$l_2^{WORST} = \hat{t_2} + tCL$ $l_2^{BEST} = tCL$	OP: Different Columns and RR/WW	$tRCD + tBUS$
$l_2^{WORST} = \hat{t_2} + tWL$ $l_2^{BEST} = tWL$	OP: Different Columns and RW	$tRCD + tBUS + tRTW$
$l_2^{WORST} = \hat{t_2} + tRL$ $l_2^{BEST} = tRL$	OP: Different Columns and WR	$tRCD + tWL + tBUS + tWTR$
$l_2^{WORST} = \hat{t_2} + tRP + tRCD + tCL$	OP: Different Rows and RR/RW	$\text{MAX}(tRAS, tRCD + tRTP)$
$l_2^{BEST} = tRP + tRCD + tCL$	OP: Different Rows and WW/WR	$\text{MAX}(tRRD, tRCD + tWL + tBUS + tWTR)$
$l_2^{WORST} = \hat{t_2} + tRCD + tCL$	CP: Same Bank and Rank and RR/RW	$\text{MAX}(tRC, tRCD + tRTP + tRP)$
$l_2^{BEST} = tRCD + tCL$	CP: Same Bank and Rank and WW/WR	$\text{MAX}(tRC, tRCD + tWL + tBUS + tWR + tRP)$

constraint between the CAS commands. As a result, the latency of pr_2 is $l_2^{1a} = tRCD + tWL$, which includes the minimum time between W2 and start of data transfer.

Case 2a: When $t_2 < \hat{t}$, $tRRD$ constraint is unsatisfied; hence, the MC delays A2 by $\hat{t} - t_2$ resulting in an access latency of $l_2^{2a} = (\hat{t} - t_2) + tRCD + tWL$. Now, suppose that $tBUS + tRTW > tRRD$ such that $\hat{t} = tBUS + tRTW$. There are again two cases based on the arrival time of pr_2.

Case 1b: When $t_2 \geq \hat{t}$, then $t_2 \geq tBUS + tRTW > tRRD$ and A2 can be issued immediately. When $tRCD$ cycles elapse after A2, $t_2 + tRCD \geq tRCD + tBUS + tRTW$. Therefore, CAS2 satisfies the $tBUS + tRTW$ constraint after CAS1 and can be issued resulting in a latency of $l_2^{1b} = tRCD + tCL$.

Case 2b: When $t_2 < \hat{t}$, the $tBUS + tRTW$ constraint is unsatisfied; hence, the MC delays CAS2 by $\hat{t} - t_2$ resulting in an access latency of $l_2^{2b} = (\hat{t} - t_2) + tRCD + tCL$. Since $l_2^{1a} < l_2^{2a}$

and $l_2^{1b} < l_2^{2b}$, the arrival time for pr_2 producing the best-case latency occurs when $t_2 \geq \hat{t}$ with $\hat{t} = \text{MAX}(tRRD, tBUS + tRTW)$. □

Corollary 5.3. *The latency of pr_2 at any given arrival time t_2 when $\hat{t_2} = \text{MAX}(tRRD, tBUS + tRTW)$ is given by:*

$$l_2 = \text{MAX}(\hat{t_2} - t_2, 0) + tRCD + tWL.$$

When $t_2 \geq \hat{t_2}$ in Corollary 5.2 will give the best-case latency $l_2^{BEST} = tRCD + tWL$, while substituting $t_2 = 0$ will give the worst-case latency $l_2^{WORST} = \hat{t_2} + tRCD + tWL$. Similarly, we calculate the best and worst-case latency suffered by any request accessing the DRAM as well as the arrival times that cause these latencies. Table 5.1 tabulates these latencies.

5.4 Reverse-Engineering Properties of the MC

The best- and worst-case latencies presented in Section 5.3 allow us to reverse-engineer properties of the MC. We refer to open- and close-page policies by OP and CP respectively. We also refer to channels, ranks, banks, rows and columns by chn, rk, bk, rw and cl respectively. We perform a step-by-step procedure to reverse-engineer MC properties. Figure 5.5 illustrates this procedure. We first reverse-engineer the page policy. Based on the page-policy, we reverse-engineer the address mapping implemented by the MC. Finally, using knowledge about both page policy and address mapping, we reverse-engineer the command arbitration scheme. In the reverse-engineering process, we use the latency bounds illustrated by Figures 5.6a and 5.6b. Figure 5.6a presents l_2 bounds for the case of two read requests, while Figure 5.6b presents the l_2 bounds for a write followed by a read. b_j and c_j in Figures 5.6a and 5.6b represent the best- and worst-case bounds for different sequences.

5.4.1 Reverse-engineering page policy and address mapping

We use two tests to reverse-engineer both the page policy, and the address mapping. The first test performs two consecutive reads, while the second test consists of a write request followed by a read request.

Both these tests are a sequence of two logical requests, lr_1 followed by lr_2 (which the MC will translate to pr_1 and pr_2, respectively) as shown in Algorithm 5.1. The function flipBit(*addr*, *bitPos*) takes as input a logical/physical address (*addr*) and a bit position (*bitPos*), and returns a logical/physical address that differs from the input logical/physical address by a single bit position defined by *bitPos*. Therefore, logical address la_2 differs from la_1 by a single bit position

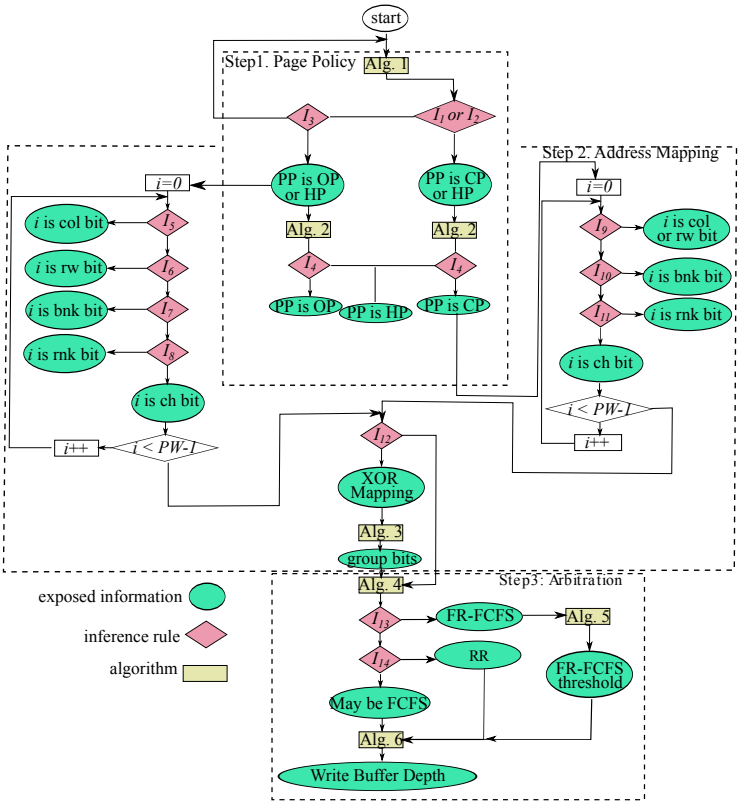
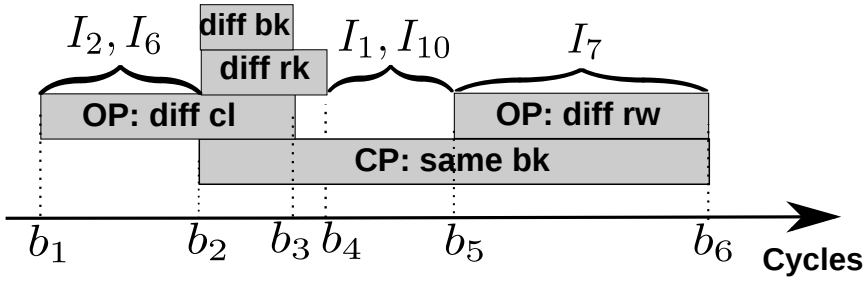


Figure 5.5: Reverse-engineering process.

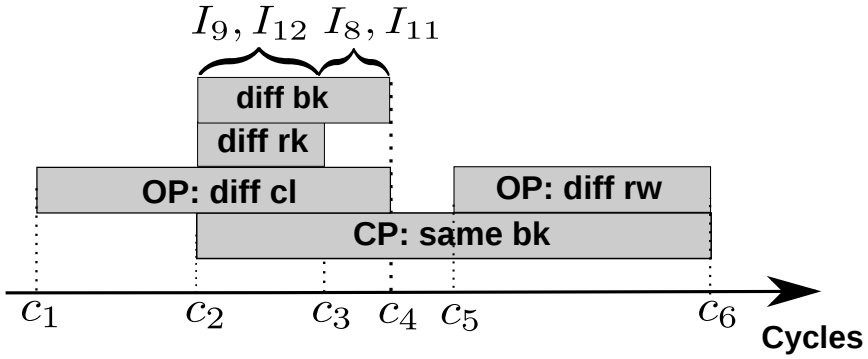
(i^{th} bit). Recall that we use the best-case and worst-case latencies to reverse-engineer the MC properties. In order to achieve such latencies, some time delay is necessary between the arrival times of the memory requests to the MC. We achieve this delay by inserting a number of NOP instructions between the requests (insertNOPs() function). We execute the tests and record the observed latencies. We repeat this for PW number of times to record the latencies observed for each bit of the logical address. Based on the latency analysis, we present inference rules for reverse-engineering the page policy and address mapping.

5.4.1.1 Step 1: Reverse-engineering the page policy

We denote the latency of the second request with the i^{th} bit flipped as l_2^i. Then, the following inference rules reverse-engineer the page policy.



(a) Latency bounds for a sequence with two consecutive reads ($test_1$). $b_1 = tCL$, $b_2 = tRCD+tCL$, $b_3 = tRCD+tCL+tBUS$, $b_4 = tRCD+tCL+tBUS+tRTRS$, $b_5 = tRP + tRCD + tCL$, and $b_6 = tRC + tRCD + tCL$.



(b) Latency bounds for a sequence of two requests: write followed by read ($test_2$). $c_1 = tCL$, $c_2 = tRCD+tCL$, $c_3 = tRCD+tCL+tBUS+tRTRS$, $c_4 = tRCD+tCL+tWL+tBUS+tWTR$, $c_5 = tRP+tRCD+tCL$ and $c_6 = tRCD + tWL + tBUS + tWR + tRP + tRCD + tCL$.

Figure 5.6: Latency bounds for a sequence of two requests.

$$(I_1)\ \exists i \in [0, PW-1] : b_4 < l_2^i < b_5 \quad \Rightarrow \textit{close-page}$$
$$(I_2)\ \exists i \in [0, PW-1] : b_1 \leq l_2^i < b_2 \quad \Rightarrow \textit{open-page}$$
$$(I_3)\ \forall i \in [0, PW-1] : l_2^i = b_2 \quad \Rightarrow \textit{close-page}$$

It is clear from Figure 5.6a that the ranges used in I_1 and I_2 do not overlap with any other range. Therefore, we can reverse-engineer the page policy. If the observed latencies do not satisfy the conditions of I_1, I_2, and I_3, we repeat the tests with different number of NOP instructions. One key observation we make from Figures 5.6a and 5.6b is that the close-page policy has a fixed

Algorithm 5.1: Reverse-engineering page policy and address mapping.

```
1  forall i in [0, PW − 1] do
2      Let test_1 = [lr_1 = ⟨la_1, R⟩, insertNOPs(),
3                    lr_2 = ⟨flipBit(la_1, i), R⟩]
4      Let test_2 = [lr_1 = ⟨la_1, W⟩, insertNOPs(),
5                    lr_2 = ⟨flipBit(la_1, i), R⟩]
6      resetMC();
7      runTest(test_1);
8      resetMC();
9      runTest(test_2);
10 end
```

best-case latency. I_3 states that if the observed l_2 is fixed for all bits and equal to $tRCD + tCL$ cycles, then the page policy is close-page. This case happens when the second request always arrives after $\hat{t_2}$ for all cases.

Hybrid-page policy. A MC implementing a hybrid-page policy dynamically adapts to either close-page or open-page behaviour based on the access pattern in order to maintain a standard of performance [111]. In order to detect hybrid-page policy implementations, an additional test is necessary, which is shown in Algorithm 5.2. $test_3$ is a sequence of $2n$ requests where the first n requests target different rows to the same bank and rank, and the last n requests target different columns to the same row, bank, and rank. n is a sufficiently large number to influence the row-hit and miss counters that are checked by the MC to adjust the page policy. As Figure 5.5 shows, we execute Algorithm 5.2 after having an initial decision on the page policy whether it is open- or close-page. If the MC implements a hybrid-page policy, then on executing the first n requests of $test_3$, the MC gradually adapts to close-page policy to reduce the DRAM access latency as they target different rows to the same bank. On the other hand, the MC adapts from close-page policy to open-page policy on executing the next n requests of $test_3$ to reduce the access latency of the requests targeting the same row. From Figure 5.6a, it is observed that in a MC implementing a close-page policy, the minimum access latency of a request is b_2. On the other hand, in a MC implementing open-page policy, the minimum access latency of a request targeting a row different from the row opened in the row buffer is b_5. Hence, if there exists an observed latency that is below b_5 in the first phase of the test, while there is an observed latency that is below b_2 in the second phase, we can deduce that the page-policy implemented is hybrid-page policy. Inference rule I_4 is based on these observations for reverse-engineering hybrid-page

Algorithm 5.2: Reverse-engineering hybrid-page policy.

Let $lr_k = \langle la_k, R \rangle, k \in [1, n]$
where:
$(bnk_l = bnk_m) \wedge (rw_l \neq rw_m), \forall l \; \forall m \in [1, n]$
Let $lr_j = \langle la_j, R \rangle$, $j \in [n+1, 2n]$
where:
$(bnk_l = bnk_m) \wedge (rw_l = rw_m), \forall l, \forall m \in [n+1, 2n]$
Let $test_3 = [lr_1, \text{insertNOPs}(), ..., lr_n, \text{insertNOPs}(),$
$lr_{n+1}, \text{insertNOPs}(), ..., lr_{2n}]$
resetMC();
runTest($test_3$);

policy.

$$(I_4)\ \exists k \in [1, n], \exists j \in [n+1, 2n] : (l_k < b_5) \wedge (l_j < b_2) \Rightarrow \textit{hybrid-page}$$

5.4.1.2 Step 2: Reverse-engineering the address mapping

Open-page or Hybrid-page. Assuming that the page policy inferred is open-page or hybrid-page, the address mapping scheme is reverse-engineered in the following way.

1. Column and row bits: It can be observed from Figure 5.6a that the access latency range on executing $test_1$ for column and row bits do not overlap, resulting in the following inferences.

$$(I_5)\ \forall i \in [0, PW-1] : b_1 \leq l_2^i < b_2 \qquad \Rightarrow i \text{ is a column bit.}$$
$$(I_6)\ \forall i \in [0, PW-1] : b_5 \leq l_2^i \leq b_6 \qquad \Rightarrow i \text{ is a row bit.}$$

2. Rank, bank and channel bits: Rank and bank bits are inferred using $test_2$ of Algorithm 5.1. A write followed by a read request that target different banks to the same rank causes the MC to reverse the direction of the shared data bus. This switching overhead distinguishes the worst-cast access latencies of requests targeting different banks in the same rank from those targeting different ranks. Inference rules I_7 and I_8 are based on this observation. The channel bits are simply the remaining bits.

$$(I_7)\ \forall i \in [0, PW-1] :\ (i \text{ is not a column bit}) \wedge (c_3 < l_2^i \leq c_4) \qquad \Rightarrow i \text{ is a bank bit.}$$

$$(I_8)\ \forall i \in [0, PW-1] :\ (i \text{ is not a column or bank bit}) \wedge (c_2 < l_2^i < c_3) \quad \Rightarrow i \text{ is a rank bit.}$$

Close-page. Suppose the MC implements close-page policy.

1. Column and row bits: From Figure 5.6a, the access latency range between b_4 and b_5 is unique to close-page policy and moreover, unique to either a row or column access under close-page policy. Inference rule I_9 uses this observation to reverse-engineer the row or column bits.

$$(I_9)\ \forall i \in [0, PW-1] : b_4 < l_2^i < b_5 \Rightarrow i \text{ is a row or column bit.}$$

   This inference implies that under close-page it is not possible to distinguish between row and column bits if the address mapping scheme places them successively. For instance, the row and column bits cannot be distinguished for the following address mapping scheme $\langle chn, rw, cl, rk, bk \rangle$. However, they are distinguishable for the following address mapping scheme $\langle chn, rw, rk, bk, cl \rangle$. This is because, in general, the number of row bits and the number of column bits are different, thus distinguishable if they are not successive to each other, while the bits of each segment (row or column) are contiguous.

2. Rank, bank and channel bits: We use $test_2$ to reverse-engineer the rank, bank and channel bits for the reason explained in the open-page policy as depicted using inference rules I_{10} and I_{11}. The remaining bits are the channel bits.

$$(I_{10})\ \forall i \in [0, PW-1] : (i \text{ is not a column or row bit}) \wedge (c_3 < l_2^i \leq c_4) \Rightarrow i \text{ is a bank bit.}$$

$$(I_{11})\ \forall i \in [0, PW-1] : (i \text{ is not a column, row or bank bit}) \wedge (c_2 < l_2^i < c_3) \Rightarrow i \text{ is a rank bit.}$$

XOR address mapping. To reduce high access latencies for requests targeting different rows to the same bank, some modern MCs employ XOR bank interleaving [112–114] to convert some of the requests targeting different rows to the same bank to requests targeting different banks. XOR bank interleaving is achieved by performing an XOR operation between the bank bits and an equivalent number of row bits. This results in more bank bits exhibiting similar access latencies on executing $test_1$ and $test_2$ of Algorithm 5.1. Since the number of bits assigned to each group (channel, rank, bank, row and column) is known from the specifications, the following inference rule detects an XOR address mapping. Initial bank bits (IBB) refer to the bits detected by inference rule I_7 or I_{10} as bank bits.

For a MC with XOR address mapping between bank and row bits, it is not possible to detect which bits of IBB actually map to the bank bits. This is because all IBB bits result in the same latency. Nonetheless, it is possible to classify IBB into two groups. One of these groups is the bank bits and the other one is the row bits; however, without being able to decide which group represents the bank bits. Algorithm 5.3 achieves this classification by issuing two requests, lr_1

and lr_2. lr_2 differs from lr_1 in only two bits out of IBB, i and j. As Inference I_{13} highlights, if the latency of the second request, $l_2^{i,j}$ is such that the two requests have a row conflict, then i and j belong to two different groups, i.e. one of them is a bank bit and the other one is a row bit. The intuition behind this decision is that $XOR(i,j) = XOR(\bar{i},\bar{j})$. Accordingly, flipping i and j results in mapping lr_2 to the same bank; hence, these two bits are XORed together by the MC. Table 5.2 further illustrates this conclusion by showing some examples.

$(I_{12})\ \forall i \in [0, PW-1]:\ \text{IBB} \geq BKW \Rightarrow$ ***XOR mapping***

$(I_{13})\ \forall i,j \in \text{IBB}:\ l_2^{i,j} > b_3 \Rightarrow (i \text{ is a bank bit} \wedge j \text{ is a row bit}) \vee (i \text{ is a row bit} \wedge j \text{ is a bank bit})$

Algorithm 5.3: Reverse-engineering XOR address mapping.

forall i, j ***in*** *IBB* **do**
  Let $test_1 = [lr_1 = \langle la_1, R\rangle$, insertNOPs(),
    $lr_2 = \langle$flipBit(flipBit(la_1, i), j), $R\rangle]$
  where $j \neq i$
  resetMC();
  runTest($test_1$);
  resetMC();
end

5.4.2 Reverse-engineering the command arbitration scheme

Based on the page policy and address mapping scheme inferred from steps 1 and 2, we reverse-engineer three common arbitration schemes *First-In-First-Out (FIFO)*, *Round Robin (RR)* and *First-Ready-First-Come-First-Serve (FR-FCFS)* using the following procedure. Algorithm 5.4 uses two tests denoted as $test_5$ and $test_6$ to reverse-engineer the arbitration scheme. In $test_5$, lr_1 and lr_3 target the same rank, bank, and row, and lr_2 targets a different row to the same rank and bank. In $test_6$, lr_1 and lr_2 target different rows to the same rank and bank, and lr_3 targets the same rank, but a different bank. These tests are designed based on the characteristics of the above mentioned command arbitration schemes, and can be inferred based on the reordering of data returned by the MC due to these requests. We execute the tests, and record f_1, f_2 and f_3.

Table 5.2: Results of XORing different bank and row bits.

req	original XORed		new		*access pattern*
	row	bank	row	bank	
r_1	101	001	101	100	*Row conflict*
r_2	100	000	101	100	
r_1	101	001	101	100	*Different bank*
r_2	111	000	111	111	
r_1	101	001	101	100	*Different bank*
r_2	101	010	101	111	

1. FR-FCFS and RR: We use the results from the tests to define the following inference rules.

   (I_{14}) When using $test_5$, $(f_3 < f_2) \Rightarrow$ *scheme is FR-FCFS.*
   (I_{15}) When using $test_6$, $(f_3 < f_2) \Rightarrow$ *scheme is RR.*

   I_{14} states that if the data transfer for lr_3 begins before that of lr_2, then the MC implements FR-FCFS. This is because FR-FCFS favours requests accessing the open row. I_{15} indicates that $f_3 < f_2$ happens when the MC selects lr_3 over lr_2 after servicing lr_1 because the MC grants access to a request accessing a bank that is different than that of lr_1's. This shows that the MC implements RR between banks.

2. FIFO: If the observed finish times are in FIFO order $f_1 < f_2 < f_3$, then either the MC implements FIFO arbitration scheme or the requests arrive to the MC command queue such that the command for the next request arrives after the first command of the previous request is issued. Therefore, in order to reverse-engineer FIFO command arbitration scheme correctly, the requests have to access the MC such that request lr_3 arrives to the MC before the issuance of lr_2's first command, and no re-orderings are observed in both tests.

5.4.3 Advanced MC features

5.4.3.1 FR-FCFS threshold

FR-FCFS arbitration scheme prioritizes ready requests (row buffer hits) over non-ready requests (requests that target different rows). This prioritization decreases the average-case access latency to the DRAM; nonetheless, it starves the non-ready requests. Therefore, MCs often enforce a hardware threshold to bound the number of prioritized ready requests that are consecutively

Algorithm 5.4: Reverse-engineering arbitration schemes.

Let $lr_1 = \langle la_1, o_1 \rangle$,
    $lr_2 = \langle la_2, o_2 \rangle$,
    $lr_3 = \langle la_3, o_3 \rangle$
Let $test_5 = [lr_1, \mathsf{insertNOPs}(), lr_2, \mathsf{insertNOPs}(), lr_3]$
where:
$(bnk_1 = bnk_2 = bnk_3) \wedge (rw_1 = rw_3 \neq rw_2)$
Let $test_6 = [lr_1, \mathsf{insertNOPs}(), lr_2, \mathsf{insertNOPs}(), lr_3]$
where:
$(bnk_1 = bnk_2 \neq bnk_3) \wedge (rw_1 \neq rw_2 \neq rw_3)$
resetMC();
runTest($test_5$);
resetMC();
runTest($test_6$);

serviced. On achieving this threshold, a P command is sent to close the row in the row buffer. We introduce Algorithm 5.5 to reverse-engineer this threshold. We issue a sequence of RDY requests that target the same rank, bank, and row, and increment RDY until a request with latency $l_2 \geq b_2$ is observed. This occurs only when the row buffer is precharged by the MC due to reaching the threshold set by the arbitration scheme on the number of row buffer hits to be serviced. Hence, the number of requests serviced before this latency is inferred as the FR-FCFS threshold.

5.4.3.2 Write buffer

Since read accesses are more latency sensitive than write accesses, MCs usually prioritize reads over writes [28]. This is deployed by queuing write accesses in a write buffer and by designating a threshold for the maximum possible number of writes backlogged in the buffer. If the number of writes in the buffer is less than this designated threshold, read accesses can be serviced before write accesses given that the system's memory ordering model is preserved. It is important to reverse-engineer if the MC has a write buffer and the size of this buffer, if there is one, because it will affect the worst-case latency for different request types. For example, the worst-case access latency of a write request will increase as read requests will have higher priority. We propose Algorithm 5.6 to perform this reverse-engineering. We issue a read request, lr_1, followed by a sequence of WQ write requests and finally another read request, lr_{WQ+1}. If WQ is less than the buffer threshold, then the last read request, lr_{WQ+1}, will be serviced before the first write request

Algorithm 5.5: Reverse-engineering FR-FCFS threshold depth.

Let $RDY = 2$ be a counter.
repeat
  Let $lr_i = \langle la_i, R \rangle, \forall i \in [1, RDY]$
  Let $test_7 = [lr_1,$ insertNOPs(), $lr_2,$
    insertNOPs(), ..., $lr_{RDY}]$
  where:
  $(bnk_1 = bnk_i) \wedge (rw_1 = rw_i), \forall i \in [2, RDY]$
  resetMC();
  runTest($test_7$);
  `inc` (RDY);
until $(\exists\ l_2 : l_2 \geq b_2)$

in the buffer, lr_2. We repeat the test and each time increment WQ by 1. When the MC services lr_2 before lr_{WQ+1}, we infer that WQ is equal to the write buffer threshold.

5.4.4 Performance Counters

In order to reverse-engineer the architecture of MCs, we require specific performance counters. Although there exist certain MC performance monitoring units (PMUs) on conventional architectures such as the Intel Xeon and Intel Core i7 platforms [115], we believe that these are insufficient for reverse-engineering the architectures of MC. This is because existing PMUs count the number of a specific type of MC command such as A, CAS, CASp, P, and REF, and do not capture the time-stamp at which these commands are issued. Therefore, in order to accurately reverse-engineer the MC, we assume that the considered platform has performance counters that can track the time-stamps of the requests when they access the MC and are retired by the MC in real-time.

5.5 Potential Applications

The reverse-engineering procedure we propose exposes implementation details of the main memory system. In this section, we discuss possible applications that can leverage this information.

Architecture Simulators. Architecture simulators are prominently used to validate and eval-

Algorithm 5.6: Reverse-engineering write buffer depth.

Let $WQ = 2$.
repeat
  Let $lr_1 = \langle la_1, R \rangle$,
    $lr_i = \langle la_i, W \rangle \forall i \in [2, WQ]$,
    $lr_{WQ+1} = \langle la_{WQ+1}, R \rangle$ such that:
  $test_8 = [lr_1, \mathsf{insertNOPs}(), lr_2,$
    $\mathsf{insertNOPs}(), .., lr_{WQ+1}]$
  where $\forall i, j \in [1, WQ + 1]$:
  $bnk_i = bnk_j$, $rw_i = rw_j$, and $cl_i \neq cl_j$
  resetMC();
  runTest($test_8$);
  `inc` (WQ);
until $(l_{WQ+1} > l_2)$

uate novel policies and compare different approaches. These simulators require a detailed model of the hardware architecture to be simulated. There exist a set of DRAM simulators that model state-of-the-art main memory systems [110, 116, 117]. However, these simulators resemble a subset of the MC policies, merely because the implementation details of other policies are not publicly available. Accordingly, the more MC policies that can be reverse-engineered, the more comprehensive and accurate the simulators will become.

Compiler and Source-code Optimizations. Memory latency is a critical bottleneck to achieve higher performance in modern computing systems. Many approaches addressed this issue by proposing compiler techniques and source-code modifications to increase the memory performance [96,97]. The main idea behind these approaches is to layout the data footprint of the application to increase data locality. Locality is a key factor for memory performance since data blocks that are close to each other are accessed faster. This is true for caches (e.g. requests to same cache line), and for DRAMs (e.g. requests targeting same row). However, if the implementation details of the memory hierarchy are not available, exploiting these locality opportunities is limited. For instance, the aforementioned approaches [96, 97] focus only on array structures, since they are placed contiguously in the memory by default.

WCET Analysis. Real-time platforms consist of memory hierarchies with a combination of on-chip scratchpads, caches, and DRAMs [93]. It is imperative for the memory hierarchy to be predictable to allow WCET analysis tools to account for latencies incurred during memory accesses. Unfortunately, manufacturers consider the implementation details of the memory hierarchy as intellectual property; hence, this information is not publicly available neither for

caches [95] nor for DRAMs [98]. Consequently, WCET analysis tools have to consider conservative models of the architecture of the memory hierarchy. Unfortunately, this leads to pessimistic WCET estimates [29, 95]. Exposing the architecture details of the memory hierarchy will definitely lead to more realistic and tight WCET estimates for real-time systems platforms.

Hardware Attacks. Exploiting the architecture details of the memory system creates new vulnerabilities in the memory system, which open the door for hardware attacks. Examples for these attacks include: Denial-of-Service(DoS) [118], Error Injection [101], Covert- [119], and Side-channel attacks [99]. Addressing these vulnerabilities is crucial to guarantee system's security; though, it is not the focus of this thesis.

5.6 Experimental Evaluation

We use the proposed algorithms and inference rules in Section 5.4 to reverse-engineer the MC properties of the Xilinx Virtex-5 based XUPV5-LX110T development board with an on-board DDR2 memory module.

We implement the algorithms into a testbench that executes on the board's synthesized MC. The Xilinx development board enables us to prove the ability of the proposed methodology to reverse-engineer a real commercial platform; however, since its MC deploys a pre-defined subset of policies, it does not allow for exploring all the capabilities of the proposed methodology. Therefore, in addition to the board, we use a simulation framework consists of macsim [39], an X86 simulator integrated with DRAMSim2 [110], a comprehensive DRAM simulator. We extend DRAMSim2 to integrate state-of-the-art MC's policies to illustrate the capabilities of the proposed methodology to reverse-engineer them in a multi-core architecture executing high-level programs. Section 5.6.1 discusses our findings for the Xilinx board, while Section 5.6.2 discusses the findings on the simulation framework.

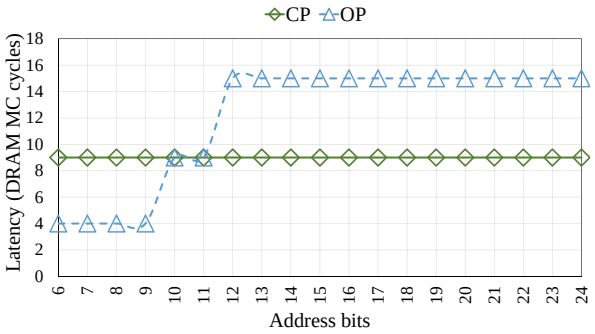
5.6.1 Reverse-engineering MC's properties of the XUPV5-LX110T platform

5.6.1.1 System specifications

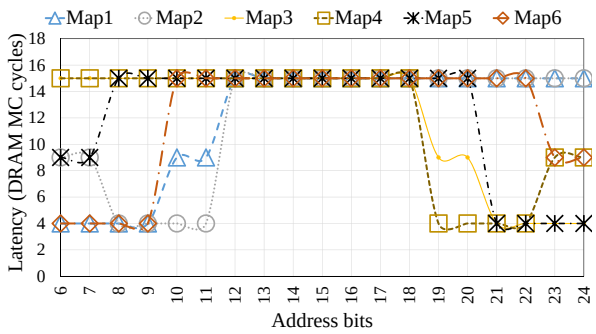
The Xilinx XUPV5-LX110T development system [120] integrates a 256MByte DDR2 memory [121]. The memory module is organized as a single memory rank with 16 data bits per column. The memory controller is generated and synthesized on the board using the memory interface generator (MIG) [122], which is part of the Xilinx toolchain. Table 5.3 tabulates the timing

Table 5.3: DDR2 specifications.

Property	Value
# of row bits	13
# of column bits	10
# of bank bits	2
Address mapping scheme	All possible permutations of $pa = \langle bnk, rw, cl \rangle$ (default is $rw : bnk : cl$)
Page policy	Open-page (default) or close-page policy
Command arbitration scheme	FIFO
DDR2 operating frequency	267 MHz
Timing constraints in cycles	tCL=4, tRCD=4, tRP=4, tRTP=3, tRAS=11, tRRD=3



(a) Default mapping (Map1) and different page policies.



(b) Default page-policy (OP) and different address mappings.

Figure 5.7: Latency plots for $test_1$ stimulating the on-board MC of XUPV5-LX110T. OP: open-page policy, and CP: close-page policy.

constraints of the DDR2 memory deployed on the board, and the properties of the MC.

5.6.1.2 Methodology

The XUPV5-LX110T board allows us to configure the memory controller. Accordingly, we configure the memory controller with certain properties. Afterwards, we test if the proposed methodology can figure them out. We integrate the reverse-engineering algorithms into a synthesizable testbench that is executed on the on-board MC. The testbench generates a sequence of read and write requests to the MC. These commands are queued into a 32 entry wide FIFO buffer, from which the MC issues to the DDR2 memory. The board provides a real-time probing of MC's interface signals through a tool called ChipScope Pro. We use ChipScope Pro to moni-

Table 5.4: Address mapping findings for XUPV5-LX110T.

Map.	cl bits	bnk bits	rw bits	
Map1	$[6, 9]$	$[10, 11]$	$[12, 24]$	$rw : bnk : cl$
Map2	$[8, 11]$	$[6, 7]$	$[12, 24]$	$rw : cl : bnk$
Map3	$[21, 24]$	$[19, 20]$	$[6, 18]$	$cl : bnk : rw$
Map4	$[19, 22]$	$[23, 24]$	$[6, 18]$	$bnk : cl : rw$
Map5	$[21, 24]$	$[6, 7]$	$[8, 20]$	$cl : rw : bnk$
Map6	$[6, 9]$	$[23, 24]$	$[10, 22]$	$bnk : rw : cl$
Inference	I_5	$b_2 \leq l_i \leq b_3$	I_6	

tor the arrival and finish time of memory requests as defined in Section 5.3. Recall that $test_2$ in Algorithm 5.1 helps in distinguishing between rank and bank bits. Since the on-board DRAM has a single rank, we execute only $test_1$ and we do not need to run $test_2$.

5.6.1.3 Results

Figure 5.7 delineates the obtained results upon executing $test_1$. Given that the page policy and the address mapping of the MC are configurable, we first use the default address mapping and run $test_1$ twice, once with the page policy configured as open-page and the other with close-page. The observed latencies of these two executions are shown in Figure 5.7a. The CP configuration in Figure 5.7a has a fixed latency for all bits. From Inference I_3, we deduce that it has a close-page policy. On the other hand, bits $[6, 9]$ of the OP configuration has a latency of 4, which is less than $tRCD + tCL$. Accordingly, using Inference I_2, we deduce that it has an open-page policy. Afterwards, to show the ability of the proposed methodology to disclose the address mapping bits, we use the MC default settings except for the address mapping, where we experiment with all allowable permutations of the board. Figure 5.7b illustrates the monitored latencies for running $test_1$ on each mapping. Table 5.4 summarizes our findings.

To reverse-engineer the command arbitration scheme, we deploy $test_5$ and $test_6$ and observe the data bus signal. We observe that the order of requests issued by the on-board MC and the order of data blocks returned by the memory module are identical. Using inferences I_{13} and I_{14}, we infer that the command arbitration scheme implemented by the on-board MC is likely a FIFO. Note that we confirm the correctness of our inferences against reference design guides, and inspecting the synthesizable MC code design if necessary.

5.6.2 Evaluation on simulation framework

5.6.2.1 System specifications

The simulation framework services two purposes in our evaluation. First, it provides a multi-core architecture executing real C programs. For the Xilinx board, a fine-grained access to the MC is possible through testbench generators; hence, we can provide our algorithms in the form of a trace of read and write accesses. Clearly, this situation is not possible in more complex commercial-of-the-shelf (COTS) systems. Accordingly, the architecture simulator provides an emulation for these systems. Second, as aforementioned, the simulator is flexible to modify; thus, we extend it by integrating a wide set of state-of-the-art MC policies. Tables 4.1 (in Section 4.2) and 5.5 show the DRAM memory module specifications and processor configurations, respectively. We deploy the targeted policies into three MC configurations: MC A, MC B, and MC C. Table 5.6 tabulates the properties of each MC.

Table 5.5: System configuration.

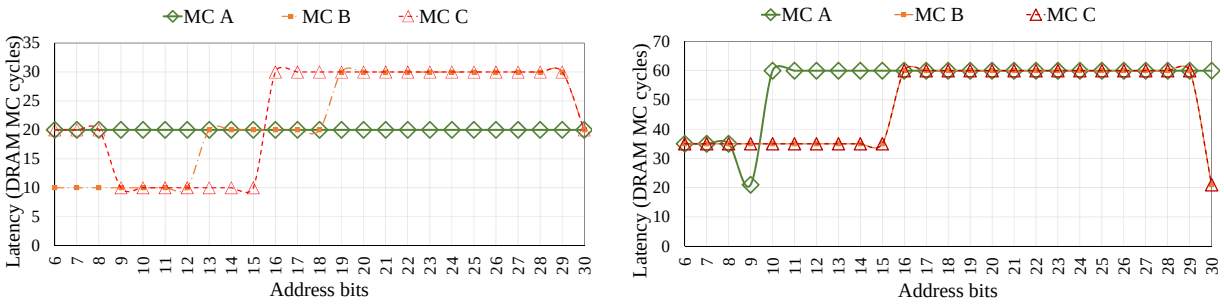
Parameter	Configuration
Core specifications	3 GHz, 5 stages out-of-order pipeline, 256-entry reorder buffer
Cache specifications	L1 I-cache: 4 KB, 8-way 8-set, 64B line size L1 D-cache: 16 KB, 4-way 64-set, 64B line size L2 D-cache: 32 KB, 8-way 64-set, 64B line size
DRAM specifications	Single channel, 1600 MHz DDR3, 64-bit data bus BL=8, 2 ranks, 8 banks per rank, 16 KB row buffer size

5.6.2.2 Methodology

We implement the reverse-engineering algorithms as micro-benchmarks in C with inline assembly code, which are executed by the processing core. We compile the micro-benchmarks with no optimization flags to ensure that the reverse-engineering requests are not optimized in any way that might change the order of requests accessing the DRAM. We execute the memory requests intended for reverse-engineering the MC for a sufficiently large number of iterations in order to offset the effects of DRAM refreshes, and elicit stable latencies of the requests intended for reverse-engineering. In addition, we have to ensure that these requests access the DRAM and are not fetched from the cache. This can be achieved by multiple ways. Modern architectures enables bypassing the cache hierarchy through special instructions. For instance, the x86 ISA provides bypass instructions for reads/writes with no temporal locality [123]. The proposed reverse-engineering methodology also works for architectures that do not support cache bypass-

Table 5.6: MC configurations.

Parameter	MC A	MC B	MC C
Address Mapping	*chn*:*rw*:*cl*:*rk*:*bk*	*chn*:*rk*:*rw*:*bk*:*cl*	*chn*:*rk*:*rw*:*cl*:*bk*
Page-policy	Close-Page	Open-Page	Hybrid
Arbitration	Round-Robin	FR-FCFS	FIFO



(a) Latency plots for $test_1$

(b) Latency plots for $test_2$

Figure 5.8: Latency plots for page policy and address mapping inference tests.

ing. For those architectures, we execute a sequence of memory instructions based on the cache hierarchy on each iteration such that the reverse-engineering requests are evicted from the cache. We rely on the methods proposed in [95] to determine the cache structure and replacement policy to generate the cache eviction requests. After ensuring that the reverse-engineering requests access the DRAM, we execute the reverse-engineering program and measure the latencies of these requests. Afterwards, we apply the inference rules proposed in Section 5.4 on these measured latencies to reverse-engineer the MC properties.

Recall that the algorithms specified in Section 5.4 insert NOP instructions between the requests intended for reverse-engineering to achieve specific access latency ranges necessary to reverse-engineer the properties of the MC. In addition, we also execute a number of NOP instructions after performing the cache evict requests in order to ensure that they are completed, and no requests occupy the store buffer or instruction buffer. The number of NOPs used is determined based on the frequency scaling factor between the core and MC, the length of instruction/reorder buffer, and cache miss penalties. In order to avoid any reordering of reverse-engineering requests by the requestor, we create data dependencies between the reverse-engineering requests.

5.6.2.3 Results

Page policy and address mapping. We first reverse-engineer the address mapping and page policy implemented by the MC. Figures 5.8a and 5.8b show the access latencies for different MCs on executing $test_1$ and $test_2$ of Algorithm 5.1 respectively. On executing $test_1$, MCs B and C implement an open-page policy from inference rule I_2 as bits 6-12 and bits 9-15 exhibit latency equivalent to $tCL = 10$ cycles. Applying inference rule I_3, MC A implements a close-page policy as all the bits have the same latency ($tRCD + tCL$ cycles) on executing $test_1$. The column and row bits for the MCs implementing an open-page policy are inferred based on inference rules I_5 and I_6 respectively. The column bits for MCs B and C are bits 6-12 and 9-15 respectively, and the row bits are bits 19-29 and bits 16-29 respectively. The bank and rank bits for all the MCs are identified by executing $test_2$, and applying inference rules I_7 and I_8 for open-page policy and I_{10} and I_{11} for close-page policy respectively. From Figure 5.8b, the bank bits for MCs A, B, and C are bits bits 6-8, 13-18, and 6-8 respectively. Note that for MC B, the bank bits are 13-18 and row bits are 19-29. This contradicts the DRAM memory module specifications of 8 banks or 3 bank bits, and 16K rows or 14 row bits. Applying inference rule I_{12}, the 6 bank bits can be inferred as an XOR combination of the three bank bits and three lower significant row bits.

Hybrid-page policy. Figure 5.9 shows the latencies of the reverse-engineering requests for one iteration on executing $test_3$ of Algorithm 5.2 for MC C. Note that we execute $test_3$ for all MC configurations, and show only the MC configuration that exhibited latencies aligning with the latencies in inference rule I_4. Recall that $test_3$ executes a sequence of n requests targeting different rows in the same bank followed by another sequence of n requests targeting the same row. It is observed from Figure 5.9 that some of the initial accesses targeting different rows to the same rank and bank incur a precharge overhead resulting in an access latency of $tRP + tRCD +$

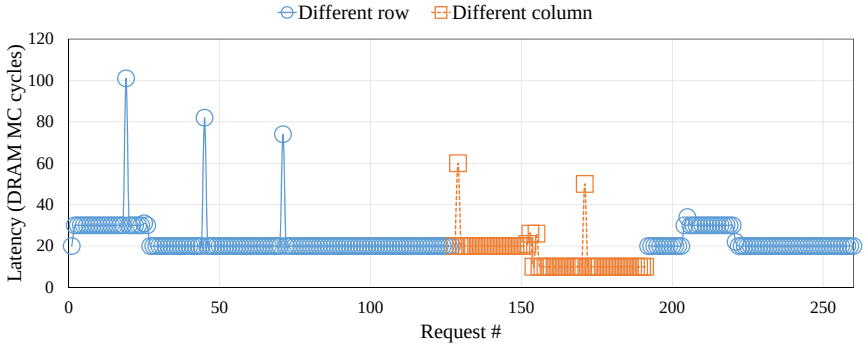


Figure 5.9: Latency plot for hybrid-page policy.

tCL cycles (30 cycles). However, the page policy adapts to the incoming access sequence and precharges the row buffer soon after the previous request has completed its operation resulting in subsequent accesses targeting idle row buffers. This is observed in the change in latency for accesses targeting different rows to the same rank and bank from $tRP + tRCD + tCL$ to $tRCD + tCL$ cycles (20 cycles). On executing the next access sequence that target different columns to the same row, bank, and rank, the latency for the requests remains at $tRCD + tCL$ cycles as the current state of the hybrid-page policy precharges the row buffer soon after a request has completed it operation. Therefore, despite accesses targeting different columns to the same rank, bank, and row, each access incurs the latency of activating the row buffer. Again, the hybrid-page policy adapts to favour the row buffer hits by delaying the precharge to the row buffer after each access. This is observed in the latency change for requests in the second access sequence from $tRCD + tCL$ to tCL cycles (10 cycles). On repeating these two sequences, as Figure 5.9 shows, MC adapts between close- and open-page policies to reduce the DRAM access latency. Notice that some requests have access latencies higher than the possible access latency, which is $b_6 = 54$ cycles. It is likely that these requests arrived when the MC was refreshing the DRAM banks, and therefore stalled until the refresh completed.

Arbitration Scheme. We execute $test_5$ and $test_6$ and infer the command arbitration scheme by comparing the order of data returned by the MC with the request order issued to the MC. For MC A, we observe the same order of requests issued and data returned for $test_5$, but a different order of data returned and request order for $test_6$. In $test_6$, the first and second request target different rows to the same bank and rank, and the third request targets a different bank. We notice that the third request to a bank different from the first two requests of $test_6$ completes earlier than the second request.

From inference I_{14}, we can infer that the command arbitration scheme in MC A is RR. For MC B, we observe a different order between the returned data and the issued requests for $test_5$. Recall, that $test_5$ generates three requests to the same bank, with the first and third request targeting the same row, and the second request targeting a different row. We observe that the third request is serviced before the second request, which indicates that the MC prioritizes requests to rows present in the row buffer. This observation aligns with the inference rule I_{13}, and hence MC B implements FR-FCFS command arbitration scheme. For MC C, the order of data returned by the MC and the request issue order are the same for both tests. Hence, we infer that the command arbitration scheme implemented in MC C is FIFO.

FR-FCFS threshold. $test_7$ in Algorithm 5.5 exposes the threshold enforced by the MC to limit the number of row buffer hits before pre-charging the row buffer for FR-FCFS arbitration scheme. The latency plot for this test is shown in Figure 5.10. From Figure 5.10, the threshold enforced by the MC is 4 as after every 4 accesses to the same row, bank, and rank, the row buffer is pre-charged. This results in every $n * 4 + 1^{th}$ request to incur the penalty of re-activating the

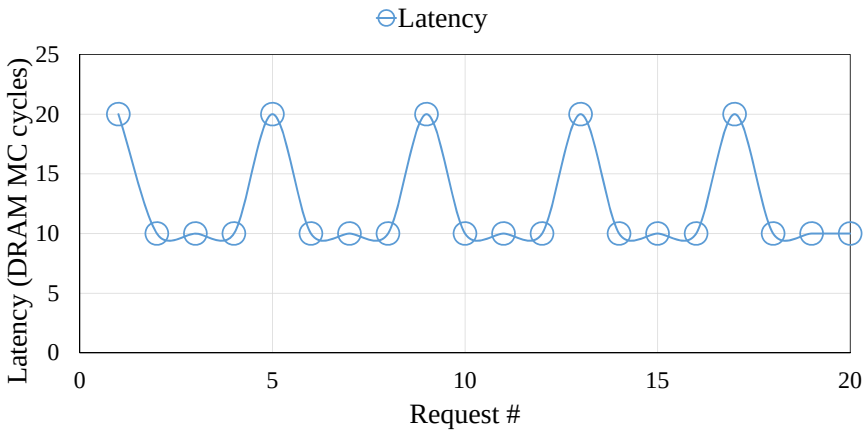
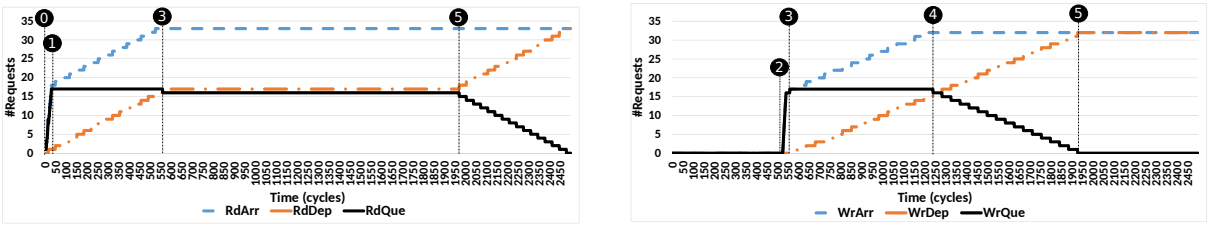


Figure 5.10: Latency plot for FR-FCFS threshold test.

row buffer. **Write buffer policy**. As aforestated, contemporary MCs favor reads over writes because read instructions stall the pipeline, while write instructions do not. They buffer writes in a separate buffer and schedule them according to various policies. In order to show the ability to demystify the write buffer information, we integrate the following write management policy in DRAMSim2. We split the unified transaction queue into two separate queues, one for reads and one for writes. The write queue has two watermarks: a high watermark, HI, and a low watermark, LO. When the write queue size, WQ, exceeds the high watermark, the MC services writes until the size drops to the lower watermark. In addition, writes are serviced when there are no pending reads. We experimented with a wide variety of values for the watermarks and the write queue size. For clarity, we show only the results for the case where $WQ = HI = 16$, and $LO = 0$. In addition, we set the read queue size to 16. In this case, the MC waits for the write queue to fill up before it starts servicing writes. afterwards, it services writes until the emptiness of the write queue.

Figure 5.11a shows the histogram of the number of arriving (RdArr), departing (RdDep), and queued (rdQue) read requests. Figure 5.11b presents the same data but for the write requests. We calculate the number of queued requested of one type as the difference between the arriving and departing requests of that type, rdQue $=$ rdArr $-$ rdDep. Clearly, from both figures, we can infer that both the write queue size and the read queue size are 16. This is because the maximum value of rdQue and wrQue is 17, which represents the case of 16 queued requests, while an extra request is being serviced.

We start counting time at stamp ⓿, where the first read request arrives. Since the arrival rate of read requests is higher than the service rate, the read queue fills up quickly at stamp ❶. The first write request does not arrive until stamp ❷ at cycle 526. At stamp ❸, the write queue fills



(a) Read requests histogram.

(b) Write requests histogram.

Figure 5.11: Write buffer policy.

up; hence, the MC switches to service writes. As a result, we observe at Figure 5.11a, the RdDep plot saturates at 17. The 18^{th} read request waits until stamp ❺ to get serviced. At stamp ❹, all the write requests in the program have arrived; hence, WrQue starts decreasing until all writes are being serviced at stamp 5. At 5, the MC switches back to service requests from the read queue.

5.7 Summary

In this chapter, we investigated opportunities to reverse-engineer properties of DRAM MCs using latency-based analysis. The analysis provided us with the best and worst-case bounds on access requests to the DRAM, on which we based our inference rules for reverse-engineering MC properties such as the page policy, address mapping scheme, and command arbitration scheme. We implemented our algorithms for reverse-engineering these properties into a software tool, and our experimental evaluation confirmed that we can discover the targeted properties of the MCs. We also showed the effectiveness of the proposed approach by reverse-engineering the implementation details of the MC deployed in the XUPV5-LX110T platform from Xilinx. Finally, we highlighted possible applications that can exploit the discovered MC policies.

Chapter 6

MCXplore: Automating the Validation Process of DRAM Memory Controller Designs

This chapter presents an automated framework for the validation of DRAM MCs called MCXplore. In developing this framework, we construct formal models for memory requests and DRAM command interaction. The framework enables validation engineers to define their test plans precisely using temporal logic specifications. We use the NuSMV model-checker to generate counterexamples that serve as test templates. MCXplore uses these test templates to generate memory tests to validate the correctness properties of the memory controller. We show the effectiveness of MCXplore by validating various state-of-the-art MC features as well as hard-to-detect timing violations. We also provide a set of predefined test plans, and regression test suites that validate essential properties of modern DRAM MCs. We release MCXplore as an open-source framework to allow validation engineers and researchers to extend and use.

6.1 Introduction

While the complexity of computing systems is increasing, their time-to-market is decreasing. As a consequence, the validation process of computing systems becomes a major challenge that consumes a considerable portion of the design cycle. Companies spend millions of dollars annually on the validation process of all components of the computing system [124]. Researchers have proposed methodologies to validate CPU designs [125, 126]. However, with the increase in memory requirement demands from applications, main memory subsystem is a vital component

in almost all computing systems. Therefore, the validation of the memory subsystem is as crucial as validating other components. Thus, this chapter focuses specifically on validating the MC component, which manages requests to the main memory.

There are several techniques to validate computing systems. We consider simulation-based validation since it is a commonly used approach [125, 126]. To validate any new feature or debug failures in the memory subsystem using the simulation-based approach, validation engineers adopt a simulation model. They provide stimulus inputs to the model and study its responses. Consequently, the effectiveness of this approach is heavily dependent on the ability of input tests to cover necessary execution scenarios to be validated. Different approaches exist for generating these tests. The straightforward approach is to use available benchmarks as the input stimuli, which saves time and cost required to develop test suites. This approach is extensively used by researchers to evaluate and validate their novel MC designs [75, 114, 127–129]; though, it has shortcomings. First, some of the benchmarks may not be memory intensive. Furthermore, they may be so complex that they do not have easy-to-analyse memory patterns, which are vital to diagnose MC responses and to check for correctness. Second, these benchmarks do not explore the state space of the memory subsystem properties. For instance, they have specific locality and read/write switching ratios. Exploring this state space is paramount for validating the design under all possible scenarios. To avoid these shortcomings, validation engineers either manually develop their own synthetic test suites or use random test generators [124] [125]. Manually-generated tests are time consuming and prone to human errors. On the other hand, randomly-generated tests may not cover all necessary test properties.

A MC is a complex component that has to track the state of all memory banks, check over 12 timing constraints obligated by the JEDEC standard [11], and make several dynamic arbitration decisions. Having the memory as a bottleneck in many computing systems [94], they are becoming even more complex with different performance optimizations such as multiple reordering levels, adaptive policies, and priority-based arbitration. Therefore, test generation for memory subsystem validation is becoming an increasing challenge.

6.1.1 Contributions

We address the aforementioned challenge by making the following contributions.

1. We present MCXplore, an automated framework for the validation of MCs. Unlike prior efforts to validate MCs, MCXplore is design independent such that it can be used to validate any MC design. This is possible because MCXplore instead of following the convention by modeling a specific MC design, it models the input stimulus, which is common across different MC implementations. Further, since MCXplore is design independent, it can be used to validate

the MC at different design cycles such as the pre-silicon high-level modeling (e.g. using simulators), hardware register-transfer level (RTL) implementations, and post-silicon validation. MCXplore enables validation engineers to precisely specify the properties required in the test suite in temporal logic specifications. Then, it automatically generates tests with the optimal number of memory requests that satisfy these properties to validate the correctness of the MC. We release MCXplore as an open-source framework [130] to allow validation engineers and researchers to extend and use.

2. We introduce two formal models for the generation process of memory tests. The first model represents the interrelation amongst memory requests (Section 6.4.1.1) and is used to validate the MC's frontend (Section 6.5), while the second model resembles interactions between memory commands (Section 6.4.1.2) and is suitable for validating the MC's backend (Section 6.6). These models allow us to encode the test generation process as a symbolic finite state machine (FSM), and use model checking techniques to explore the state space for MC test suites and generate counterexamples that serve as test templates. MCXplore uses these test templates to generate property-driven test suites.

3. We highlight interesting sequence patterns that a test suite should encompass to test and evaluate various MC features. Consequently, we provide a set of predefined test plans as well as regression tests that validate essential functionalities of modern DRAM MCs.

4. Finally, we show a methodology to use high-level statistics such as bandwidth utilization, access latency, and aggregated number of committed commands to validate the correctness of several state-of-the-art MC features and debug for any timing violations. The validated MC features are just examples to show the capabilities of using MCXplore for validation. The proposed methodology applies for other features as well.

6.2 Background: Model Checking

Figure 6.1 delineates the basic operation of a model checker. A model checker is a verification tool that takes two inputs: 1) a system model expressed as an automaton or an FSM, and 2) a property expressed in a temporal logic statement. It, then, checks whether the system violates this property or not by exploring the state space of the model. If it detects a violation, it produces a counterexample. This counterexample is a path of states that falsifies the checked property. If the property is carefully crafted, this counterexample can be interpreted as a test case for the system [131]. Since the state space of the system may be exponential, it is useful to bound the number of searchable states by the checker. This approach is known as *bounded model checking*

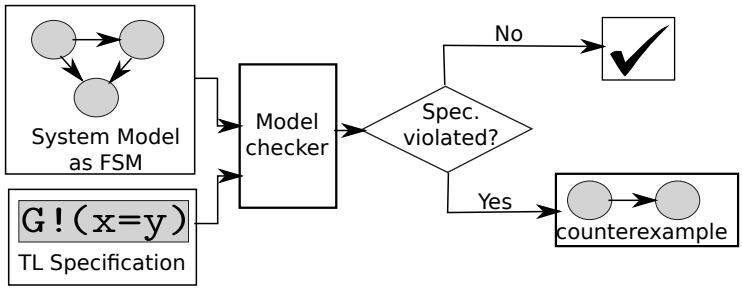


Figure 6.1: Model checking operation.

(BMC) [132]. In BMC, the checker searches for a counterexample in executions whose length is bounded by some integer i. Leveraging BMC, MCXplore provides memory tests with optimal (minimum) number of requests. Optimality is achieved by starting with $i = 1$ and increasing i until a counterexample is found. We provide more details about test generation in Section 6.4.

6.3 Related Work

Researchers have proposed several novel features to reduce the large DRAM access latency. We divide these efforts into two main categories. The first category is providing DRAM simulation environments to help in the process of evaluating the strengths and weaknesses of new ideas. Examples include DRAMsim2 [110], USIMM [117], DrSim [133], Ramulator [116], and DRAMSys [134]. The second category is proposing novel features in all memory controller sub-components such as address mapping [114, 127], page policy [3, 75, 128], and arbitration [129, 135, 136]. Validating DRAM MC designs or simulators is a challenging task for many reasons. 1) The MC must carefully track the state of each DRAM bank. 2) The DRAM JEDEC standard specifies more than 12 timing constraints that any MC must satisfy for correct operation. 3) Many dynamic factors impact the MC operation such as the type of requests in the queues and which DRAM cell their addresses target.

For these reasons, most of the proposed DRAM simulators do not fully validate their operation. Ramulator [116] is validated using 10 million memory requests that are a mix of random and hand-crafted requests. The authors conduct the validation process by providing these requests to both Ramualtor and a reference Verilog model provided by Micron [137], and comparing the behaviour of both. This approach has many shortcomings. 1) The used requests do not necessarily exhibit all the possible request and command interactions. Thus, validation coverage is not guaranteed. 2) The adopted Verilog reference model only validates that there are no violations

in the timing parameters. A timing parameter is violated if a corresponding command is issued earlier than the specified timing constraint. However, it does not test for performance penalties that may result from issuing the commands later than the specified constraint. 3) It does not validate all other policies at the MC's frontend, because these policies are not directly related to command generation. These policies include the page policy, address mapping, and request arbitration. Developers of DRAMsim2 [110] follow a similar validation procedure; thus, they suffer from the same aforementioned shortcomings. DRAMSys [134] also uses a similar approach. It executes testing scripts on the generated commands to check if the timing constraints comply to the JEDEC standard. To our knowledge, neither USIMM [117] nor DrSim [133] are validated.

In Chapter 6, we propose MCXplore to provide an easy-to-adopt methodology to validate these simulators to avoid these shortcomings. First, MCXplore provides property-driven tests for each policy that guarantees coverage of all possible behaviours, as we illustrate in Section 6.4. Second, the methodology can be used to seamlessly validate both backend and frontend policies of MCs, as we illustrate in Section 6.5. We intentionally insert bugs into several components of DRAMsim2 [110] and show that MCXplore is able to detect those bugs.

Upon proposing novel MC policies or features, researchers usually validate them using benchmarks such as in [75, 114, 127–129], or manually-written directed tests or a combination of both such as in [3, 135, 136]. Similarly, in industry, pre-silicon validation engineers often use hand-written directed tests or randomly-generated tests [124]. In Chapter 6, we propose an automated portable process of validating new features in the DRAM subsystem that can be used both by researchers and industry. Compared to the aforestated methods used in [3, 75, 114, 124, 127–129, 135, 136], our proposed framework would achieve better coverage, is less error-prone, and reduces validation complexity through automation.

6.3.1 Formal Verification of the Memory System

Some prior works incorporate formal models in the design and validation process of memory systems [138–141]. Authors of MSimDRAM [138] model their MC design as a state machine. Accordingly, they encode the correct intended behavior of that MC as linear temporal logic (LTL) specifications and use BMC to verify this behavior. Authors of [139] use a universal verification methodology (UVM) environment to verify their proposed generic MC. The environment consists of a test driver that generates test cases stimulating both the design and a reference transaction level modeling (TLM) model. The environment compares results from the design with the reference results to detect any faults. The methodology in [140] automatically transforms the timing constraints from the JEDEC standards into system verilog assertions (SVA), which can be used to verify that the commands generated by the MC comply with these constraints. A timed automata model for the MC's backend is proposed in [141]. It uses UPPAAL model checker to

verify the correct behaviour of it including the timing constraints. All these approaches target to validate a particular MC with specific design instance. The approach in [139] requires a reference TLM model to compare against and hardware probing capabilities to monitor different signals. In addition, it requires a methodology to generate representative test cases to stress the MC design. With the increasing complexity of MC designs, industry reports that the test case generation process is becoming time consuming and requires huge intellectual efforts [142]. The methodology in [140] requires modifications to the register transfer level (RTL) implementation of the MC to insert the assertions. In addition, approaches of [140] and [141] focus solely on MC's backend.

MCXplore, on the other hand, focuses on modeling the input stimulus of the MC, which enables MCXplore to be design agnostic and can be used to validate both MC's frontend and backend. MCXplore can leverage different levels of mentoring capabilities. For instance, in Sections 6.5 and 6.6, we utilize high-level metrics such as bandwidth utilization to validate features from both frontend and backend. MCXplore, similar to these works, adopts model checking in its framework. However, previous works use model checking to model and validate a specific design as in [138], or to convert the specifications into RTL assertions as in [140]. In contrast, MCXplore uses model checking as a test generation engine. Integrating model checking techniques in the test generation process is not a new idea. Model checking has proven its success as a test generation engine for validating both software [143, 144] and hardware [125, 126]. A comprehensive survey of using model checking in testing can be found in [145]. Using model checking set MCXplore as a framework that can generate various type of tests. For instance, it can generate *directed test cases*, where the generated test is based on the temporal logic specification and can be used to stress certain behavior of the MC. In addition, MCXplore can generate *constrained random test cases*, where the generated test is randomized but obeys certain rules. This is possible because model checking fully explores the state space of the memory test components (addresses, transaction types, etc). Accordingly, by specifying the rules as specifications, the model checker randomly picks one of the valid paths with minimum transitions (which can be multiple) that satisfies these rules.

Industry solutions. Unfortunately, industrial solutions are intellectual properties (IPs) with only few information available about them. Synopsys has a verification IP to verify the DRAM and the memory controller [146]. The IP is implemented in SystemVerilog and uses UVM. This IP requires access to the native RTL implementation. Further, it requires licensing, which may be of unaffordable cost. Keysight Technologies and FuturePlus Systems follow a different approach by using special probes and analyzers to monitor the DRAM signals and verify their correctness [147]. This approach is suitable after the manufacturing process is finished (post-silicon validation); however, it requires special hardware tools to conduct the verification process, which may not be costly effective. MCXplore, contrarily, can also be used for post-silicon validation,

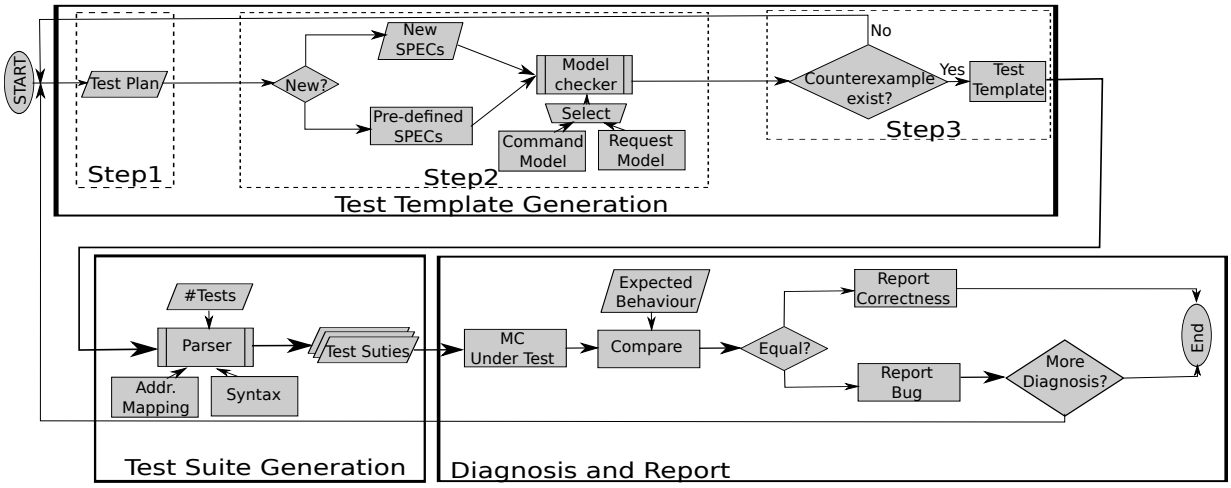


Figure 6.2: Proposed validation process of MCs.

while it does not require special hardware nor additional costs as it is freely available.

6.4 MCXplore Methodology

Figure 6.2 represents the steps of the methodology we propose. The process consists of three phases: test template generation, test suite generation, and diagnosis and reporting. Thus, the process separates the test generation step from the test plan step. This is an important requirement from validation engineers to simplify the validation process [148].

Phase** 1**: Test Template Generation– In this phase, MCXplore turns the test plan into a test template in three steps.

- *Step* 1*:* A test plan is a list of behaviours whose correctness needs to be validated. Usually, design engineers provide this list in a highly-abstracted human language.
- *Step* 2*:* The big challenge for validation engineers is to turn the test plan into meticulous rules that generated tests must follow [149]. We promote leveraging model checking capabilities to address this challenge. Model checking automates the state-space exploration of the test generation, and provides a formal methodology to define test properties. We create two abstract models to express the stimulus test of the MC: a request model, and a command interaction model. We encode them as FSMs in the NuSMV model checker [150]. Accordingly, validation engineers are able to encode test properties as specifications expressed in temporal logic

formulas. Formulas are negated such that they are true if required test properties do not exist. We accompany MCXplore with regression suites, and a pre-defined set of temporal logic specifications that encode most of the basic test properties required to stress MC designs. Table 6.1 tabulates these properties.

- *Step* 3*:* The model checker explores the FSM to determine the truth or falsity of the specifications. For a false specification, it constructs a counterexample, which is a trace of states that falsifies the specification. This trace represents the test template that encompasses test properties specified by validation engineers. We use BMC to obtain the trace with minimum number of states, which results in tests with the optimal (minimum) number of memory requests satisfying specified properties. Minimizing the number of requests is necessary to reduce the time and complexity of the validation process.

Phase 2***: Test Suite Generation–*** We provide a parser script to parse the test template produced by phase 1, and generate test suites with memory requests. Validation engineers drive this parser with the address mapping of the MC, number of desired tests in the suite, and test syntax.
Phase 3***: Diagnosis and Report–*** Validation engineers invoke the MC under validation with the generated test suite and compare responses with the expected behaviour. If results match, then they report correctness. Otherwise, they report their diagnosis results and conduct more detailed investigation if required. Section 6.5 provides a methodology to construct test plans with certain expected behaviours to use as a golden metric to compare the MC response.

6.4.1 Proposed Models

We propose two models that are at different granularities to facilitate the test generation process.

6.4.1.1 Request Interrelation Model

To fully cover the state-space of a test of n memory read/write requests and a 32 bit address, $2^{33} \times n$ tests are needed. Clearly, such a large number of tests is prohibitively time consuming. We argue that the important factor in the coverage is not the input stream pattern. Instead, it is the MC's response to this stream. For instance, if a request to a row rw_1 is followed by a request to rw_2 in the same bank and rank, then the MC behaviour depends on whether $rw_2 = rw_1$ or not. This is because the MC decision takes into account whether it is a row hit or a row conflict, regardless of the actual values of these rows. The same observation holds for banks, ranks and channels. Hence, a state graph constructing these relationships is sufficient to represent a model

Table 6.1: Currently supported configurations.

(a) Indexing Performance

Property	Configuration
Address pattern	Linear Random Customized
Type pattern	All reads All writes Switching Random Switching% Reads-to-writes%
Address mapping	Any
Transaction size	Any
Address length	32

(b) Customized Address

Segment	Configuration
Row	Hit (for any row) Conflict Random Locality%
Bank/ Rank/ Channel	No interleave Fully interleave Random Interleave%
Column	Same Successive Random

for the test template generation step. Based on this observation, we model the interrelation between memory requests as the Kripke structure in Model 6.1. Recall that a DRAM request has an address ($Addr$), and an operation type (ty) where the address consists of 5 segments (row, column, bank, etc.), and the type is a read or a write operation. We define the proposition e for each address segment such that $e = 1$ means that the request has the same segment as its previous request, and $e = 0$ otherwise. Similarly, if $ty = 1$, then the operation is a read, and a write otherwise. To exhibit all possible relations between successive requests, we have 64 possible states. For instance, for state s_{39}, $(s_{39}, \langle 1, 0, 0, 1, 1, 1\rangle)$ denotes a read request that targets the same channel, row and column as its previous request, while it targets different rank and bank. We also maintain a set of counters to track the address pattern such as total number of requests, row hits, and bank interleavings, which we use to encode the test specifications. Note that $\text{BIN}(x, y)$ returns the y^{th} bit of a positive integer x's binary equivalent number.

6.4.1.2 Command Interaction Model

Validation engineers can use the request interrelation model to validate properties related to timing constraints ruling command interactions. However, in this case, MCXplore requires them to find out the request patterns that expose these timing constraints. This is because, using the request interrelation model, MCXplore allows specifications to be at the request level and not the command level. Therefore, we propose the command interaction model to facilitate the valida-

Model 6.1: Kripke structure for the MC input.

$MCin = \{S_{in}, I_{in}, R_{in}, L_{in}\}$ where:
$P_{in} = \{ty, e_{rw}, e_{ch}, e_{rnk}, e_{bnk}, e_{cl}\}$
$S_{in} = \{s_i : \forall i \in [0, 63]\}$ is the set of all possible states.
$I = \{s_0\}$ is the set of initial states.
$R = \{(s_i, s_j) : \forall i, j \in [0, 63]\}$ is the transition relation between states.
$L = \{(si, \langle e_{cl}, e_{bnk}, e_{rnk}, e_{ch}, e_{rw}, ty\rangle)\}$ is the labeling function where all the sets cannot be empty sets, and
ty=BIN$(i, 0)$, e_{rw}=BIN$(i, 1)$, e_{ch}=BIN$(i, 2)$, e_{rnk}=BIN$(i, 3)$, e_{bnk}=BIN$(i, 4)$, and e_{cl}=BIN$(i, 5)$.

tion of properties related to MC command generation. This model enables validation engineers to specify the timing constraints to be validated and MCXplore automatically generates the test sequence that exercises these constraints. Figure 6.3 depicts the state graph of this model that we build based on the timing constraints imposed by the JEDEC standard [11]. The vertices represent DRAM commands and the edges represent timing constraints. For example, the time between A and a P to the same bank must be at least $tRAS$. In Section 6.6, we use this model to generate test suites for validating the correctness of command generation, and checking for any timing violations. Generally, the request model is better-suited for MC's frontend validation, while the command model well-suits the MC's backend validation. Frontend policies include request arbitration, address mapping, and page policy. MC's backend is responsible for command generation and arbitration.

In the remaining of this chapter, we show case studies on applying the proposed methodology to validate the correctness of several state-of-the-art MC policies. We use DRAMSim2 [110] with DDR3-1333 DRAM to conduct the experiments. We use the DDR3-1333 module only as an example. All the lemmas and proofs in this chapter are independent of the specific values of the timing constraints; thus, they are applicable to other DRAM modules unless otherwise specified. We also insert common design bugs in the functionality of these features to determine whether the proposed methodology can discover them. We use bandwidth utilization defined in Equation 6.1 as our metric to validate the MC features. The advantage of using Uti for validation is that it does not require an engineer to observe internals of the MC. Instead, existing inputs and outputs of the MC are sufficient. In Equation 6.1, the total DRAM access cycles consists of the data transfer cycles and the overhead due to DRAM timing constraints.

$$Uti = \frac{\text{Data transfer cycles}}{\text{Total DRAM access cycles}} \tag{6.1}$$

It is worth noting that a single metric cannot cover all design bugs. Accordingly, we show the

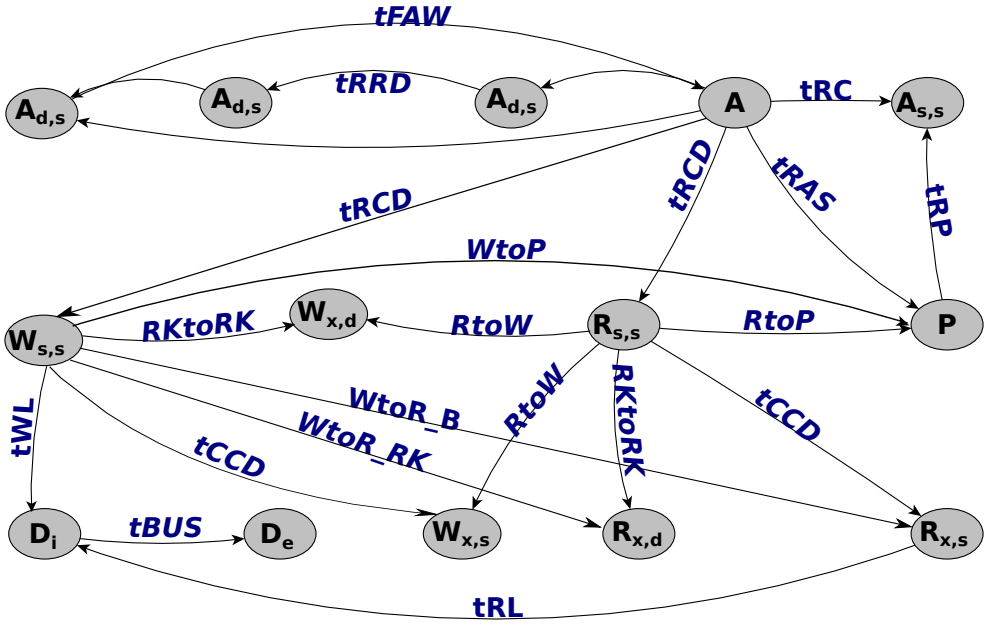


Figure 6.3: DRAM commands and timing constraints interaction. Subscripts reflect the targeted bank and rank, respectively. d: different, s: same, x: do not care. Di: start of the data transfer. De: end of the data transfer. P is for same bank. $A_{d,s}$ is an A command to a different bank on the same rank.

usage of MCXplore with other metrics such as the aggregated numbers of issued commands (Section 6.6.11), and the memory access latency (Section 6.6.12).

6.5 Validating MC's Frontend

We validate features from MCs frontend functionalities including address mapping, page policy, and request arbitration. We show the complete validation process for the XOR address mapping. For the other policies, to avoid repetition, we discuss the three major components of the validation process: the test plan, the LTL specifications, and the diagnosis. Since all the policies we validate in this section are at the MCs frontend, we use the request model.

6.5.1 Address Mapping Policies

We validate three features related to the address mapping component: the XOR address mapping [114], the address masking [151], and the rank hopping mechanism [78, 152].

6.5.1.1 Permutation-Based Page-Interleaving (XOR) Address Mapping Validation

Modern MCs reduce row conflicts by using a permutation-based page interleaving, where the bank bits are bitwise XOR-ed with the least significant three row bits [113, 114]. We refer to this technique as simply XOR address mapping.

Test plan. To generate a test suite that represents the optimal memory pattern for the XOR mapping. It is a stream of read accesses where we change the bank interleaving ratio per test, $intr$. In addition, requests targeting the same bank are accessing different rows. $Suite_{XOR}$ formally represents this test plan. Each test has an interleaving percentage between 0% and 100%. $nbnk$ is the number of banks per rank (usually 8 for DDR3). The conditions ensure that $intr\%$ of requests in the test interleave across different $nbnk$ banks. They also ensure that in these $intr\%$ requests, each $nbnk$ successive requests target same row, which implies that requests targeting different banks have same rw segment, while requests to the same bank have different rw values. Again, the target of this test plan is to achieve the maximum possible utilization of XOR mapping regardless of the $intr$ value.

Test 6.1: $Suite_{XOR}$—Test suite to validate XOR address mapping.

1 $Suite_{XOR} = \{Test_{intr} : \forall intr \in [0, 100]\}$

2 $Test_{intr} = [Req_1, Req_2, ..., Req_n]$, where $Req_k = \langle Addr_k, R\rangle, k \in [1, n]$. $((rw_l = rw_m)$ iff $((l \text{ MOD } nbnk = m \text{ MOD } nbnk) \wedge (l, m \in [1, \frac{n \times intr}{100}])))$, and $((bnk_l \neq bnk_{l-1})$ iff $l \in [2, \frac{n \times intr}{100}])$.

Specifications. Each test template has its corresponding specification. The LTL in Spec. 6.1 encodes a test plan with $intr = 40\%$, where t_x represents the total counts of the event x. `t_hit` is the total number of row hits, and `t_bank_interleave` is the total number of bank interleavings. The intuition behind the specification is that out of 10 total requests in the test, the first 6 requests target different rows in the same bank, while the last 4 requests target the same row but in different banks.

LTL Specification 6.1: XOR mapping.

```
G ((num_requests=6 & t_hit=0 & t_bank_interleave=0)->
   !F(num_requests=10 & t_reads=10 & t_hit=4 & t_bank_interleave=4))
```

Test template. MCXplore invokes the NuSMV model checker to explore the state space to find a counterexample for the specification. The counterexample represents the test template that exhibits the required test properties. Figure 6.4a delineates the test template for the specifications in Spec. 6.1.

CH1 : RNK1 : BNK1 : RW1 : CL1	R	0x00000000	R
CH1 : RNK1 : BNK1 : RW2 : CL1	R	0x00080000	R
CH1 : RNK1 : BNK1 : RW3 : CL1	R	0x00100000	R
CH1 : RNK1 : BNK1 : RW4 : CL1	R	0x00180000	R
CH1 : RNK1 : BNK1 : RW5 : CL1	R	0x00200000	R
CH1 : RNK1 : BNK1 : RW6 : CL1	R	0x00280000	R
CH1 : RNK1 : BNK2 : RW6 : CL1	R	0x00290000	R
CH1 : RNK1 : BNK3 : RW6 : CL1	R	0x002a0000	R
CH1 : RNK1 : BNK4 : RW6 : CL1	R	0x002b0000	R
CH1 : RNK1 : BNK5 : RW6 : CL1	R	0x002c0040	R

(a) Test template. (b) Final test example.

Figure 6.4: Test generation for validating XOR mapping

Test suite. MCXplore parses each test template and generates a test that complies with the test plan (step 4). Figure 6.4b shows one test instance generated from the test template.
Diagnosis. For the sake of comparison, we execute $Suite_{XOR}$ on both the XOR mapping and the base mapping (no XOR operation is performed). As Figure 6.5 illustrates, increasing the $intr$ ratio on the test, the base mapping achieves better utilization. This is because requests to different banks are serviced in parallel. On the other hand, the correct behaviour of the XOR mapping is to achieve a fixed utilization for all tests in the suite. This is because even for non-interleaved accesses, the XOR address mapping will map them to different banks because of the XOR operation between the bank bits and the corresponding row bits. To further check for correct functionality, this value should be compared to the expected utilization dictated in Lemma 6.1. Figure 6.5 shows that the XOR mapping achieves a fixed utilization of $\sim 79\%$, which coincides with the expected behaviour.

Lemma 6.1. *Executing any test in $Suite_{XOR}$ on a MC with XOR mapping results in a utilization that can be calculated as: $\frac{4 \cdot tBUS}{tFAW}$.*

Proof. Since XOR mapping maps successive requests of any test in $Suite_{XOR}$ to different banks, the MC under test repeats the behaviour shown in Figure 6.6 every 8 requests. Focusing on one repetition, the data bus is busy for $8 \cdot tBUS$, while the total DRAM latency is $2 \cdot tFAW$. □

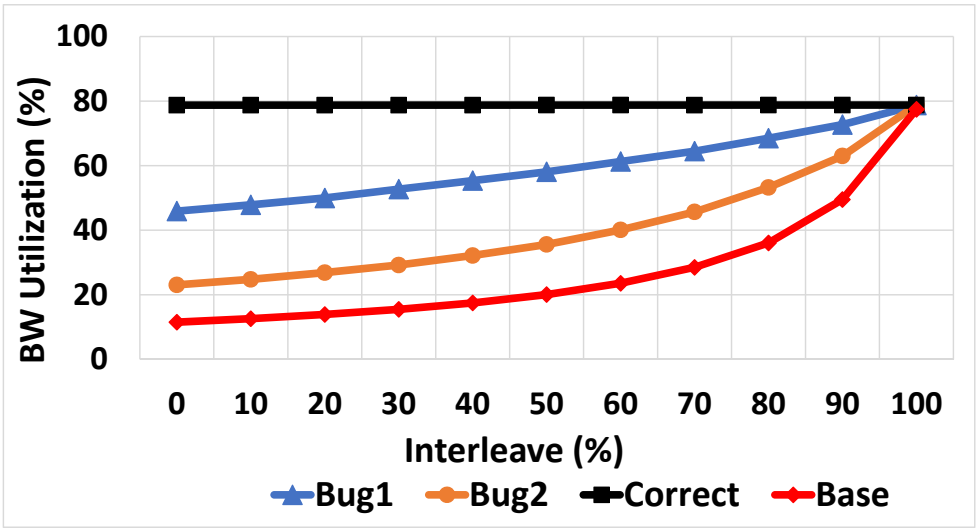


Figure 6.5: XOR address mapping.

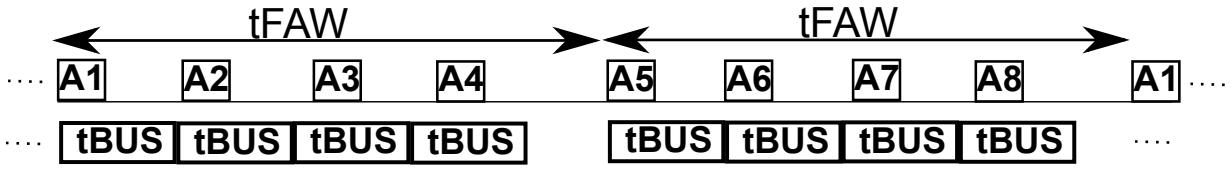


Figure 6.6: Command sequence of $Suite_{XOR}$ on XOR mapping.

Bug scenario. To illustrate potential design errors, we inject two bugs in the XOR mapping. In the first one (Bug1 in Figure 6.5), we perform the XOR operation between only the first two bits of the bank and row segments, while in the second bug (Bug2), we perform the XOR operation between the least significant bit of row and bank segments. From Figure 6.5, both Bug1 and Bug2 do not achieve the expected utilization of Lemma 6.1; hence, they are detectable.

6.5.1.2 Address Masking

MCs map logical addresses to physical addresses by using a bit masking operation. Figure 6.7 illustrates the masking operation to extract the row, column and bank segment of an address. Rank and channel segments are extracted with similar logic. We use MCXplore to generate tests to validate the address mapping operation for a variety of address mappings with different number of

rows, ranks and channels, and with various row sizes. Modern memory controllers (such as the Intel MC hub [151]) rely on configuration registers to select one of possible address mappings. For clarity, we show one simple example comprising of a single-rank single-channel DRAM subsystem with three possible address mapping schemes, $Scheme_1$: (rw-cl-bnk), $Scheme_2$: (bnk-cl-rw), and $Scheme_3$:: (bnk-rw-cl). Figure 6.8 explains these three schemes.

Test plan: To generate three tests: $Test_1$, $Test_2$ and $Test_3$. Each test is a stream of read accesses targeting the same bank and different rows. Tests differ only in the address mapping, where $Test_1$, $Test_2$ and $Test_3$ are designed corresponding to $Scheme_1$, $Scheme_2$ and $Scheme_3$, respectively.

Specification: Spec. 6.2 encodes the test plan as an LTL property, where the test comprises of 10 requests.

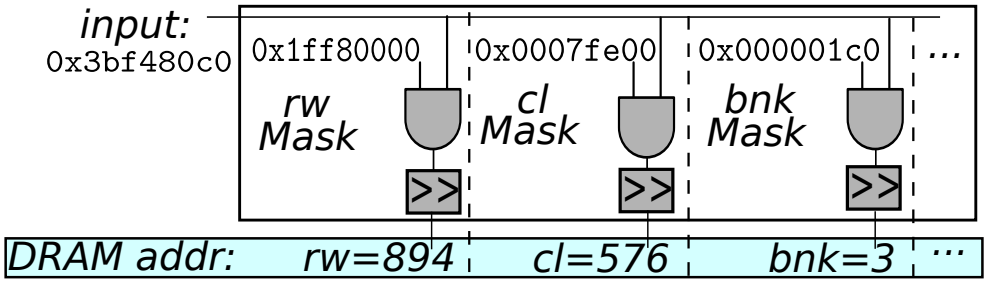


Figure 6.7: Address masking operation.

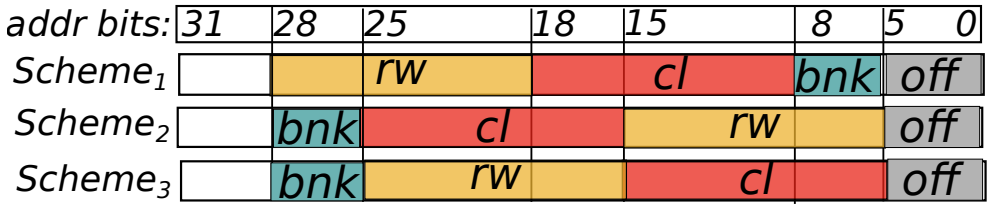


Figure 6.8: Address masking schemes.

LTL Specification 6.2: Address Masking.

```
G! ((num_requests = 10) & (t_hit=0) &  (t_bank_interleave=0))
```

Diagnosis. We execute each test on each address scheme and depict the results in Figure 6.9. Lemma 6.2 calculates the expected utilization upon executing $Test_i$ on $Scheme_i$. Each test should result in a utilization of $\frac{tBUS}{tRC} \approx 12\%$ when running on its corresponding scheme and larger utilization otherwise.

Lemma 6.2. *Executing $Test_i$ on $Scheme_i$ for $i \in \{1, 2, 3\}$, the BW utilization of the DRAM under test can be calculated as: $\frac{tBUS}{tRC}$.*

Proof. Executing $Test_i$ on $Scheme_i$ for $i \in \{1, 2, 3\}$, the DRAM under test repeats the behaviour shown in Figure 6.10 every request. Focusing on one request, the data bus is busy for $tBUS$ cycles. Moreover, the total DRAM latency required to transmit the data of one request is $tRC = tRP + tRAS$ cycles. □

Bug scenario: The masking operation may result in a different mapping than the intended one by the designer. This can be due to a fault in the masking logic, the configuration registers or

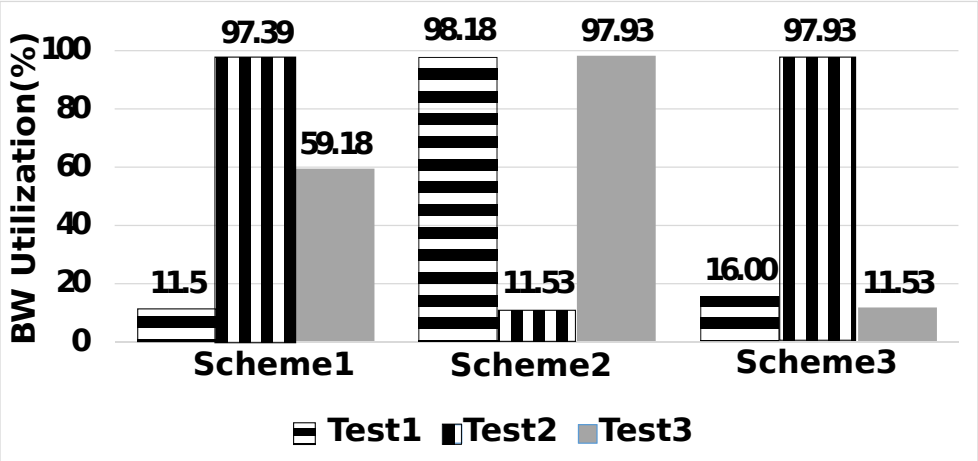


Figure 6.9: Address masking results.

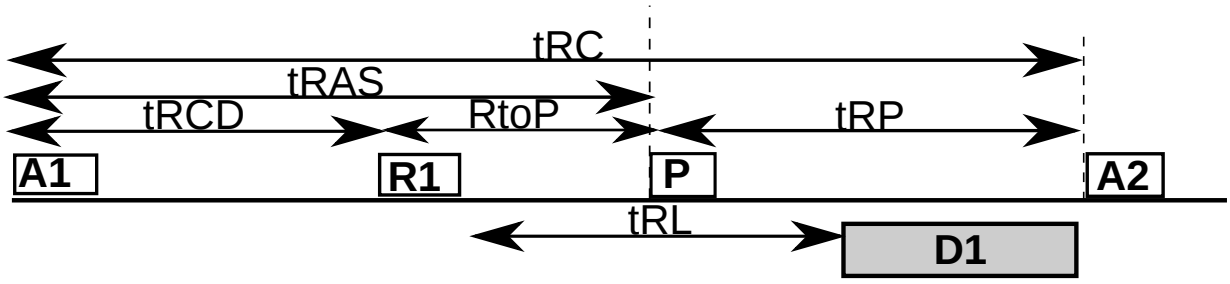


Figure 6.10: Command sequence from executing $Test_i$ on $Scheme_i$, $i \in \{1, 2, 3\}$.

even a human error when the designer unintentionally sets the wrong address scheme. For example, let the intended mapping to be $Scheme_1$, while the masking mistakenly results in a mapping of $Scheme_3$. Under this scenario, $Test_3$ is the one that will result in the expected utilization and not $Test_1$; hence, we can discover that there is inconsistency between the masking operation mapping and the intended mapping. Since the generated tests target different rows in the same bank, they generate row conflicts and result in a minimum utilization ($\frac{tBUS}{tRC}$). Accordingly, if a bug results in running $Test_i$ on $Scheme_j$, where $i \neq j$, the resulting utilization will be higher than the expected value. Results in Figure 6.9 illustrate this insight.

6.5.1.3 Rank Hopping

Some modern memory controllers use rank hopping to force consecutive requests to access different ranks [78,152]. Hence, they will not suffer from the read-write switching required between accesses of different types accessing the same rank. Instead, requests targeting different ranks suffer from a lesser delay: the rank-to rank switching, $tRTRS$.

Test plan. $Suite_{HOP}$ formally describes the test plan. The target is to generate a test suite, where each test has a different read-write switching ratio (sw) and all requests are targeting the same bank and row.

Test 6.2: $Suite_{HOP}$— Test suite for rank hopping.

1 $Suite_{HOP} = \{Test_{sw} : \forall sw \in [0, 100]\}$
2 $Test_{sw} = [Req_1, Req_2, ..., Req_n]$, where $Req_k = \langle Addr_k, ty \rangle, k \in [1, n]$. $((rw_l = rw_{l-1})$ and $(bnk_l = bnk_{l-1})\ \forall l \in [2, n])$, and $ty_l = ty_{l-1}$ iff $l \in [2, \frac{n \times sw}{100}]$

Specifications. Each test template has its corresponding specification. The LTL in Spec. 6.3 encodes a test plan with $sw = 40\%$, $(t\_sw = 40)$ out of $(num\_requests = 100)$.

LTL Specification 6.3: Rank hopping.

```
G!(num_requests=100 &t_hit=99 &t_bank_interleave=99 &
    t_sw=40)
```

Diagnosis. We test both the rank hopping mapping with a dual-rank DRAM, and a single-rank DRAM. We delineate results in Figure 6.11. The correct behaviour of rank hopping is to achieve a fixed utilization regardless the switching ratio while the utilization of the single-rank DRAM degrades with the increase of the switching ratio due to the $RtoW$ and $WtoR\_B$ constraints. For tests with low switching ratio, we expect the single-rank system to have better utilization than the rank hopping because of the overhead that $tRTRS$ constraint adds. Lemma 6.3 determines the threshold point.

Lemma 6.3. *The rank hopping mapping outperforms the single-rank base mapping if and only if the switching ratio, sw, satisfies the following condition:*

$$sw > \frac{2tRTRS}{tRL + tWTR + 2tBUS + tRTRS - 2tCCD}.$$

Proof. For the single-rank base mapping, the DRAM utilization for a test of requests with same type targeting same row and bank, $Test_{sw=0}$, can be approximated to $Uti_{no\_sw} = \frac{tBUS}{tCCD}$ (proof is in Section 6.6.1). On the other hand, the DRAM utilization for $Test_{sw=100}$ can be approximated

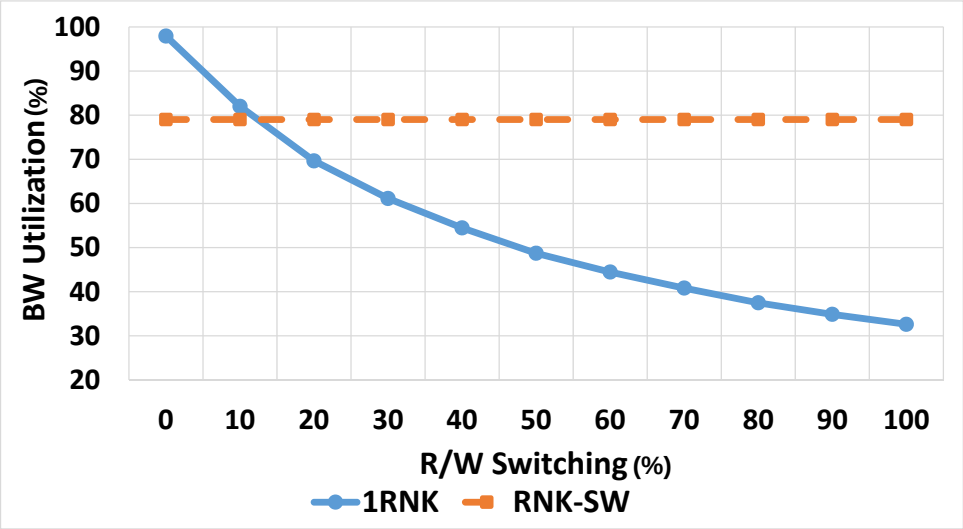


Figure 6.11: Rank hopping.

to $Uti_{sw} = \frac{2tBUS}{tRL+tWTR+2tBUS+tRTRS}$ (proof is in Section 6.6.9). Consequently, the utilization for a test $Test_{sw}$ executing on single rank is calculated in Equation 6.2. Uti_{1rnk} is the weighted harmonic mean of Uti_{no_sw} and Uti_{sw} with weights $(1 - sw)$ and sw, respectively.

$$Uti_{1rnk} = \frac{1}{\frac{1-sw}{Uti_{no\_sw}} + \frac{sw}{Uti_{sw}}} \tag{6.2}$$

For the rank hopping mapping, the DRAM utilization can be approximated to $Uti_{hop} = \frac{tBUS}{tBUS+tRTRS}$. Hence, the rank hopping outperforms base mapping if: $Uti_{hop} > \frac{1}{\frac{1-sw}{Uti_{no\_sw}} + \frac{sw}{Uti_{sw}}}$. This can occur only if $sw > \frac{Uti_{sw} \times (Uti_{no\_sw} - Uti_{hop})}{Uti_{hop} \times (Uti_{no\_sw} - Uti_{sw})}$. By substituting Uti_{no_sw}, Uti_{sw} and Uti_{hop} and conducting mathematical simplifications, the condition will be: $sw > \frac{2tRTRS}{tRL+tWTR+2tBUS+tRTRS-2tCCD}$. □

Figure 6.11 illustrates that for switching ratios that are approximately larger than or equal to 12.5%, the rank hopping mapping outperforms the single-rank base mapping which coincides with the conclusion of Lemma 6.3.

6.5.2 Page Management Policies

Recall from Section 4.2 that the page policy controls the liveness of the row in the row buffer. We validate three page policies, which are commonly used in current architectures: close-page, open-page, and adaptive-page policies. Since DRAMsim2 supports only close- and open-page policies, we extend it to support the adaptive-page policy. We implement an adaptive-page policy that models the page policy implemented by Intel [111] and executes the following procedure. If the number of row hits in a decision window is larger than 50%, the MC executes a open-page policy; otherwise, it executes a close-page policy. We choose the decision window to be 20 requests; thus, the MC executes open-page if there are at least 10 hits in the last 20 requests.

Test Plan. To generate a test suite, where each test is a stream of read accesses targeting the same bank such that it has a different locality ratio. $Suite_{PP}$ formally defines this test plan. We define the locality percentage as: $loc = \frac{\text{number of row hits} \times 100}{\text{total number of requests}}$. Unlike all previous tests, where all requests of the test are issued back-to-back at cycle 0, $Test_{loc}$ issues a request every $2 \cdot tRC$ cycles. The intuition is that it is necessary that each two successive requests be separated by a period larger than tRC to differentiate between the behaviour of the close-page and that of open-page when it has a row-conflict.

Test 6.3: $Suite_{PP}$—Test suite for page policy.

1 $Suite_{PP} = \{Test_{loc} | \forall loc \in [0, 100]\}$
2 $Test_{loc} = [Req_1, Req_2, ..., Req_n]$, where $Req_k = \langle Addr_k, R, t_k \rangle, k \in [1, n]$
3 $t_k = 2 \cdot (k-1) \cdot tRC$
4 and $(bnk_l = bnk_m)\ \forall l\ \forall m \in [1, n]$, and $(rw_l = rw_m)$ iff $l, m \in [1, \frac{n \times loc}{100}]$.

Specifications. Each loc value has its corresponding specification. The LTL in Spec. 6.4 encodes a test plan with $loc = 40\%$, $(t\_hit = 40)$ out of $(num\_requests = 100)$.

LTL Specification 6.4: Page policy.

```
G!(t_requests=100 &t_hit=40 &t_bank_interleave=0 &
    t_reads=100)
```

Diagnosis. We execute $Suite_{PP}$ on DRAMSim2 for each page policy and depict the results in Figure 6.12. 1) *Close-page* is expected to have the same DRAM utilization for all tests. Lemma 6.4 dictates this utilization. We observe that the obtained results of close-page in Figure 6.12 coincide with the expected utilization from Lemma 6.4. 2) For *open-page*, the utilization depends on the loc percentage. Increasing loc, less precharging is required and the DRAM utilization increases accordingly. Lemma 6.4 defines the relation between DRAM utilization and loc. Figure 6.12 shows that open-page policy outperforms close-page policy for $loc > 50\%$.

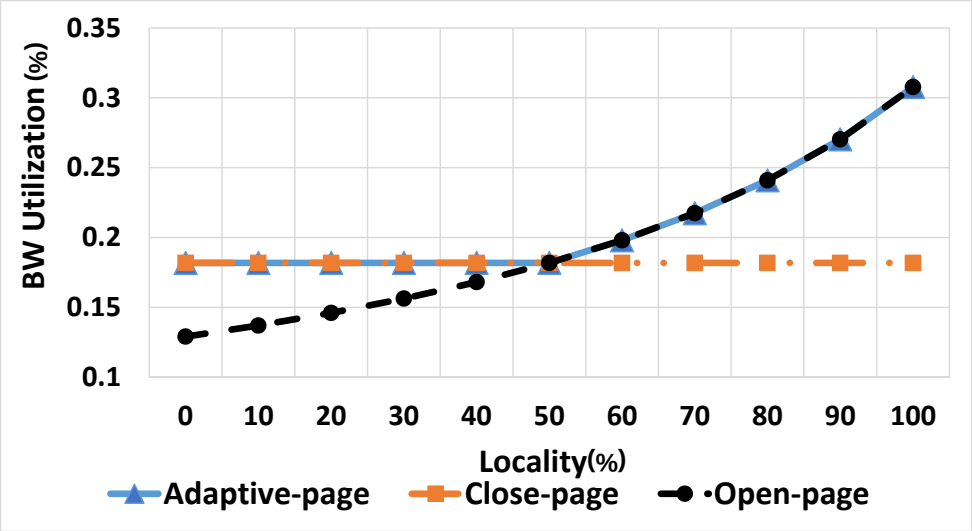


Figure 6.12: Evaluation of page policies.

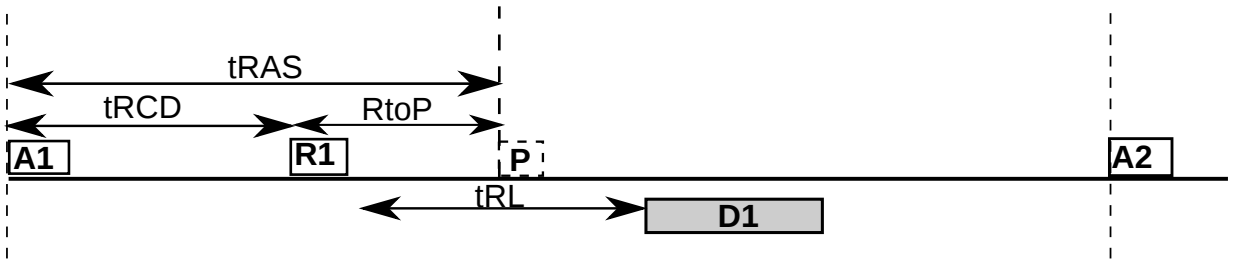
This value can be directly derived from the utilization values that Lemmas 6.4 and 6.5 calculate. 3) Deploying *adaptive-page*, the MC executes the close-page policy for low loc values, while it switches to open-page for high loc values. Figure 6.12 shows that the adaptive-page policy has the same utilization as the close-page for $loc \leq 50\%$ and the same utilization as the open-page otherwise.

Lemma 6.4. *Executing $Test_{loc}$ on a MC with close-page policy results in the same utilization for all feasible loc ratios. This utilization can be calculated as: $\frac{tBUS}{tRCD+tRL+tBUS}$.*

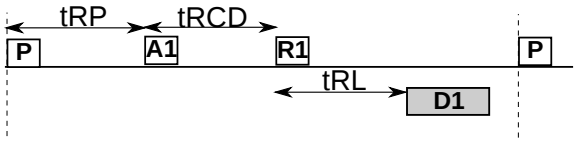
Proof. As Figure 6.13a delineates, each request consumes a total of $tRCD + tRL + tBUS$ DRAM cycles to transfer on the data bus for only $tBUS$ cycles. The DRAM remains idle for the remaining of the $2 \cdot tRC$ period and our utilization metric in Equation 6.1 only considers the DRAM active cycles. □

Lemma 6.5. *Executing any test in $Tests_{PP}$ on a MC with open-page policy, the utilization can be calculated as: $\frac{tBUS}{(1-loc)(tRP+tRCD+tRL+tBUS+loc(tRL+tBUS)}$.*

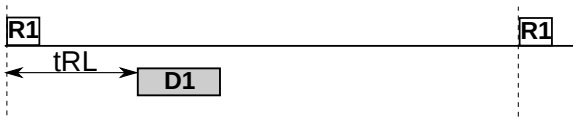
Proof. For $loc = 0$, all requests will incur a row conflict. Each request will have the command pattern in Figure 6.13b. Therefore, the BW utilization can be calculated as $Uti_{conf} = \frac{tBUS}{tRP+tRCD+tRL+tBUS}$. In contrast, for $loc = 100\%$, all requests will incur a row hit and each



(a) Close-page policy.



(b) Open-page policy (row conflict).



(c) Open-page policy (row hit).

Figure 6.13: Command arrangement for $Test_{PP}$.

request has the command pattern in Figure 6.13c. Accordingly, the BW utilization can be calculated as $Uti_{hit} = \frac{tBUS}{tRL+tBUS}$. In general, the BW utilization of $Test_{loc}$ with any loc value can be calculated as the weighted harmonic mean of Uti_{conf} and Uti_{hit} as follows: $\frac{1}{\frac{1-loc}{Uti_{conf}}+\frac{loc}{Uti_{hit}}}$. This gives the utilization value, which Lemma 6.5 calculates. □

6.5.3 Arbitration Schemes

We validate a MC feature that affects both the page policy and the arbitration deployed by the MC. MCs employing this feature keep the row in the row buffer for a designated number of row hits, that we call maximum row-hits threshold. Thus, the open-page policy is turned into a threshold-based page policy in these MCs. In addition, the threshold limits the number of requests that can be reordered with the FR-FCFS arbitration scheme deployed in most conventional MCs nowadays [136]. MC designers select the threshold value that maximizes the performance for targeted applications. In this experiment, we assume the intended threshold to be $thr = 16$. Though, the procedure is valid for any thr's value.

Test plan. To generate a set of tests, where each test is a stream of read accesses targeting the same bank, while we sweep the number of requests targeting an open row (row hits), hit, per test. We use the request model to generate the test suites. $Suite_{thr}$ formalizes this plan, where we sweep hit between 0 and 32. The conditions ensure that all requests target the same bank, while every hit successive requests target the same row.

Test 6.4: $Suite_{thr}$— Test Suite for threshold-based FR-FCFS arbitration.

1 $Suite_{thr} = \{Test_{hit} : \forall hit \in [0, 32]\}$
2 $Test_{hit} = [Req_1, Req_2, ..., Req_n]$, where (($bnk_l = bnk_{l-1}$) and (($rw_l = rw_m$) iff ($l$ MOD $hit = m$ MOD hit)) $\forall l, m \in [1, n]$).

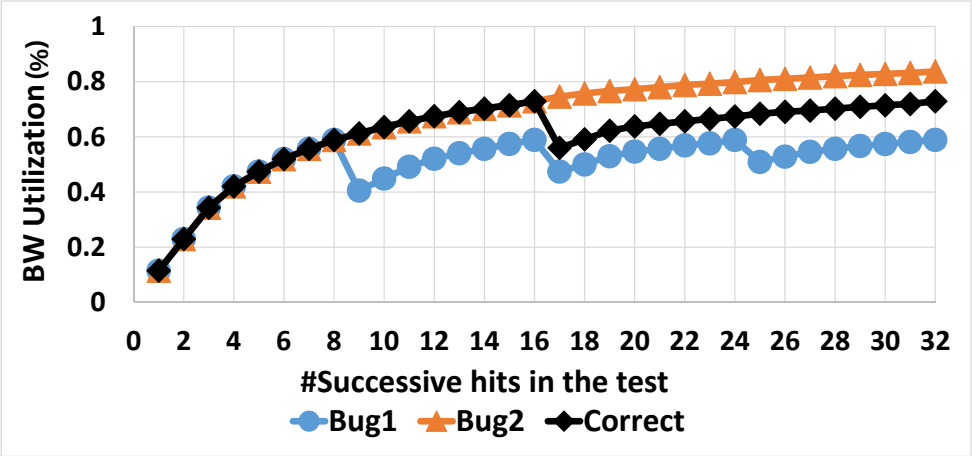


Figure 6.14: Evaluation of FR-FCFS threshold.

Specifications. The formula in Spec. 6.5 exemplifies the encoding of the test plan with $hit = 16$ such that the first request opens a row and requests 2 to 16 are hits on that row. Afterwards, request 17 opens a different row (row conflict), and requests 18 to 32 are hits on the open row.

LTL Specification 6.5: Threshold-based FR-FCFS arbitration.

```
G((c_hit=15) -> !F(t_requests = 32 & t_hit=30 &
t_bank_interleave=0 & c_hit = 15))
```

Diagnosis. We execute the generated tests and compare with the expected behaviour. The correct functionality is to achieve the maximum utilization when $hit = thr$. Lemma 6.6 calculates this maximum utilization value. Figure 6.14 shows that the MC under correct functionality (Correct) achieves a maximum utilization of 73% at $hit = 16$, which confirms the conclusion of Lemma 6.6.

Lemma 6.6. *Executing $Suite_{thr}$ on a MC that implements a maximum row-hits threshold results in a maximum utilization for the test $Test_{hit}$ with $hit = thr$ and this utilization can be calculated as:* $\frac{thr \times tBUS}{tRCD + (thr-1)tCCD + RtoP + tRP}$.

Proof. When $hit = thr$, the DRAM under test repeats the behaviour illustrated in Figure 6.15 every thr requests. During one repetition, the data bus is busy for $thr \times tBUS$, while the total access latency is $tRCD + (thr - 1)tCCD + RtoP + tRP$. □

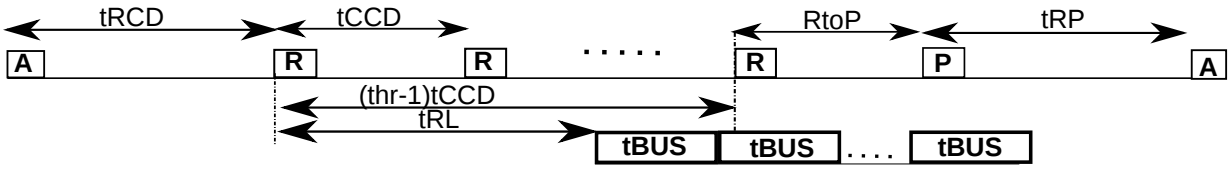


Figure 6.15: Command sequence of $Suite_{thr}$ when $hit = thr - 1$.

Bug scenario. We embed two bugs to the logic of the row-hits threshold. The first bug (Bug1 in Figure 6.14) reduces the threshold to 8 instead of the intended value by the designer (16), while the second bug (Bug2 in Figure 6.14) increases the threshold to 32. From utilization graphs in Figure 6.14, we directly discover that the maximum utilization value is not the expected value calculated by Lemma 6.6. In Bug1, the utilization graph repeats a pattern every multiple of 8, where it achieves the maximum utilization. Consequently, we deduce that the bug causes the threshold to be 8. A similar conclusion can be reached for Bug2.

6.6 Validating MC's backend

Using MCXplore, we design property-driven tests to validate the correctness of the timing parameter values enforced by the MC. The key novelty here is that each test is designed to maximize the impact of the timing parameter under test while eliminating or minimizing the effect of all other parameters. Using the state graph in Figure 6.3, we exhaustively study all possible command interactions, produce utilization equations to investigate the impact of timing parameters on utilization. Having these equations, we find that not all parameters can be isolated. As a consequence, we introduce the dependency graph in Figure 6.16. An edge from constraint $constr_1$ to $constr_2$ means that $constr_1$ must be validated before $constr_2$. A bi-directional edge between two constraints means that they have to be validated together. To avoid repetition, we only show the complete validation process for the $tCCD$ parameter. For the other parameters, we discuss the three major components of the validation process: the test plan, the LTL specifications, and the diagnosis. We summarize our findings in Table 6.4. Since we validate the timing constraints related to the command interactions, we use the command model.

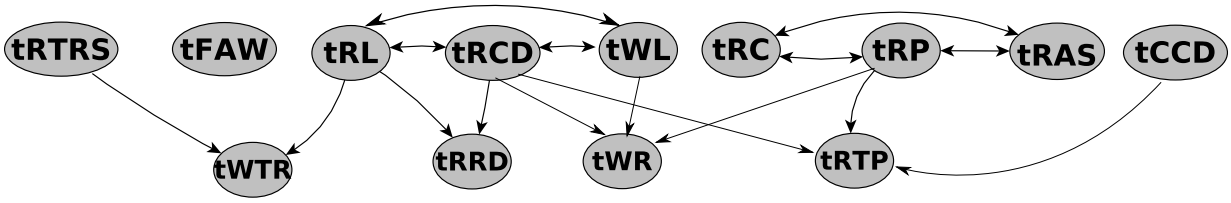


Figure 6.16: Validation dependency graph for timing parameters.

Bug scenarios. For the timing parameter under validation, we randomly set one of these parameters to a wrong value in the range: $[0, \textit{standard value} + 20]$, where *standard value* is the value dictated by the JEDEC standard.

6.6.1 tCCD

Test plan. $Test_{CCD}$ formalizes this plan. It is a stream of n read accesses targeting the same bank and row (100% row locality). If $n \gg 10$ in Figure 6.17, $tCCD$ will dominate the other timing constraints as Lemma 6.7 proves.

Test 6.5: $Test_{CCD}$ to validate $tCCD$.

$Test_{CCD} = [Req_1, Req_2, ..., Req_n]$
where $Req_k = \langle Addr_k, R \rangle, k \in [1, n]$
and $(bnk_l = bnk_m) \wedge (rw_l = rw_m), \forall l \; \forall m \in [1, n], n \gg 10$.

Specifications. The LTL in Spec.6.6 specifies a state, where the number of requests is 100 and the $tCCD$ constraint appeared 99 times on the explored path to reach this state.

LTL Specification 6.6: $tCCD$.

```
G!((num_requests=100) & (num_tCCD=99)&
((num_tRL=1)|(num_tWL=1))&(num_tBUS=1)&(num_Rx_d=0))
```

Test template and test generation. MCXplore invokes the NuSMV model checker to explore the state space to find a counterexample for the specification. The counterexample represents the test template that exhibits the required test properties. Figure 6.18a delineates the test template for the specifications in Spec. 6.6. Afterwards, the parser parses this template to generate as many tests as required, which conforms with the provided address mapping and syntax. Figure 6.18b shows one test instance for a 64bit address machine.

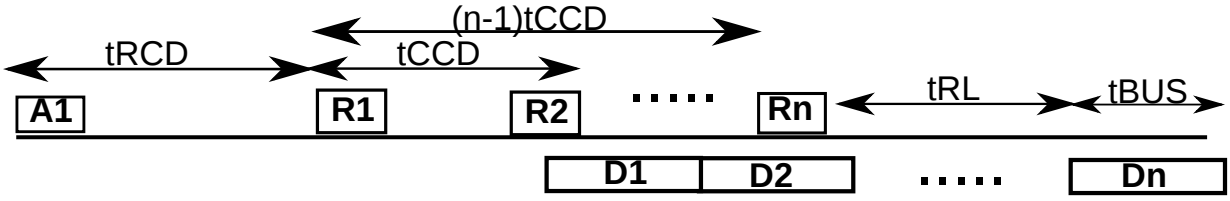


Figure 6.17: Command sequence of $Test_{CCD}$.

CH1 : RNK1 : BNK1 : RW1 : CL1	0x0000000000000040
CH1 : RNK1 : BNK1 : RW1 : CL2	0x0000000000000080
CH1 : RNK1 : BNK1 : RW1 : CL3	0x00000000000000C0
.	.
.	.
CH1 : RNK1 : BNK1 : RW1 : CL100	0x0000000000001900

(a) Test template. (b) Final test example.

Figure 6.18: Test generation for validating $tCCD$.

Diagnosis. To validate $tCCD$'s value, we execute the obtained test on the MC under test (DRAMsim2 in this case) and plot the results for various $tCCD$ values in Figure 6.19. Afterwards, we compare the observed utilization (Uti_o) with the golden-metric utilization calculated from Lemma 6.7, Uti_c. Figure 6.19 illustrates that the observed utilization aligns with the golden metric for $tCCD = 4$, which is the value specified by the standard. For $tCCD > 4$, the optimization is less than the expected value, which implies that the $tCCD$ timing is set to a non-optimal value. Obviously, this leads to a performance degradation as Figure 6.19 shows. For $tCCD < 4$, the optimization exceeds 100% in DRAMsim2 timings, which implies a corruption in the transferred data.

Lemma 6.7. *Executing $Test_{CCD}$, the BW utilization of the DRAM under test can be approximated to: $\frac{tBUS}{tCCD}$.*

Proof. Executing $Test_{CCD}$, the DRAM under test exhibits the behaviour shown in Figure 6.17. Since $Test_{CCD}$ has n requests, the data bus is busy for $n \cdot tBUS$ cycles. Using Figure 6.17, the total DRAM access time can be calculated as: $tRCD + (n-1) \cdot tCCD + tRL + tBUS$. As a result, the BW utilization of the DRAM under test is: $\frac{n \cdot tBUS}{tRCD + (n-1) \cdot tCCD + tRL + tBUS}$. If $n >> 10$, the BW utilization can be approximated to $\frac{tBUS}{tCCD}$. □

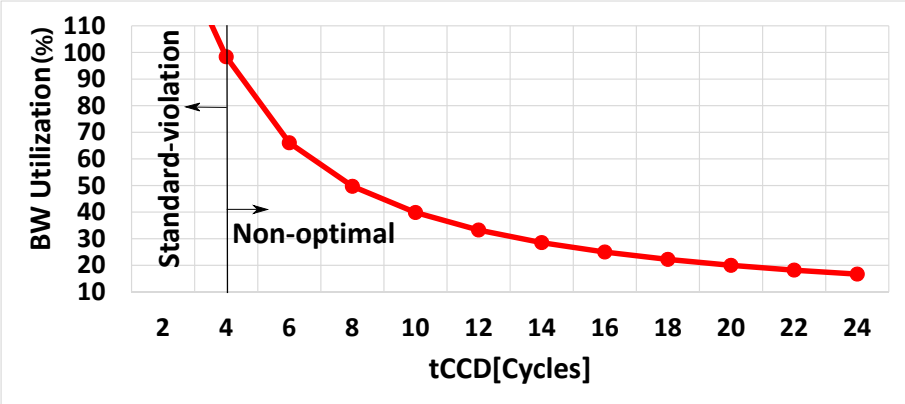


Figure 6.19: $tCCD$ validation results.

6.6.2 tRC

Test plan. $Test_{RC}$ formalizes the plan to validate tRC. It is a stream of n read accesses targeting the same bank where each access is targeting a different row than the previous access.

Test 6.6: $Test_{RC}$ to validate tRC.

1 $Test_{RC} = [Req_1, Req_2, ..., Req_n]$
2 where $Req_k = \langle Addr_k, R \rangle, k \in [1, n]$
3 and $(rnk_l = rnk_m) \wedge (bnk_l = bnk_m) \wedge (rw_l \neq rw_m), \forall l \, \forall m \in [1, n]$.

Specifications. The LTL in Spec.6.7 specifies a state in the FSM, where the number of requests is 10. The property `(num_tRC=9)` dictates that the explored path has to exhibit the tRC constraint 9 times. This ensures that all requests target a different row in the same bank.

LTL Specification 6.7: tRC.

```
G! ((num_requests = 10) & (num_tRC=9))
```

Diagnosis. Figure 6.20 depicts the observed utilization for different tRC values. Lemma 6.8 calculates the golden utilization we expect for the ideal tRC value.

Lemma 6.8. *Executing $Test_{RC}$, the BW utilization of the DRAM under test can be calculated as: $\frac{tBUS}{tRC}$.*

Proof. Executing $Test_{RC}$, the DRAM under test repeats the behaviour shown in Figure 6.10 (the same figure that illustrates command sequence of address masking) every requests. Focusing on

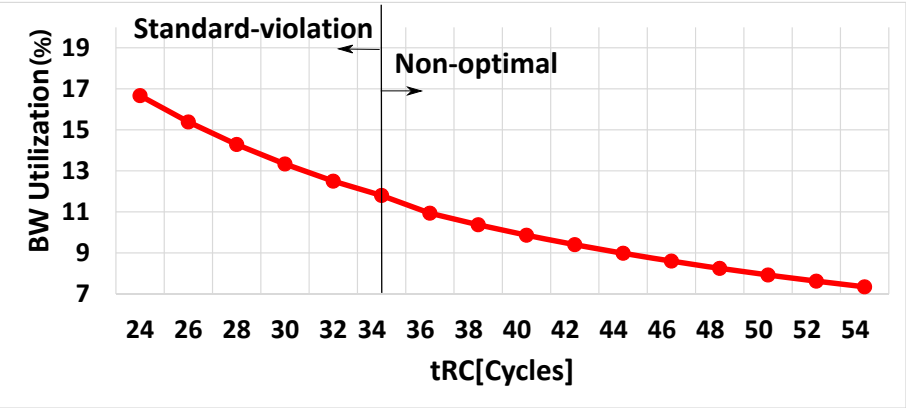


Figure 6.20: tRC validation results.

one repetition, the data bus is busy for $tBUS$ cycles. Moreover, the total DRAM latency required to transmit the data of one request is $tRC = tRP + tRAS$ cycles. □

6.6.3 tFAW

Test plan. $Test_{FAW}$ formalizes the plan to validate $tFAW$. It is a stream of n read accesses targeting the same rank where each access is targeting a different row and bank than the previous access.

Test 6.7: $Test_{FAW}$ to validate $tFAW$.

1 $Test_{FAW} = [Req_1, Req_2, ..., Req_n]$
2 where $Req_k = \langle Addr_k, R \rangle, k \in [1, n]$
3 and $(rnk_l = rnk_m) \wedge (bnk_l \neq bnk_m) \wedge (rw_l \neq rw_m), \forall l \, \forall m \in [1, n]$.

Specifications. The LTL in Spec.6.8 specifies a state in the FSM, where the number of requests is 8 and the $tFAW$ constraint appeared twice on the explored path to reach this state. A test of 8 requests is sufficient because the number of banks in the used DDR3 is 8. Recall that $tFAW$ is the minimum time to activate four banks in the same rank. If all the eight requests in the test target different banks, $tFAW$ appears twice.

LTL Specification 6.8: $tFAW$.

```
G! ((num_requests = 8) & (num_tFAW=2))
```

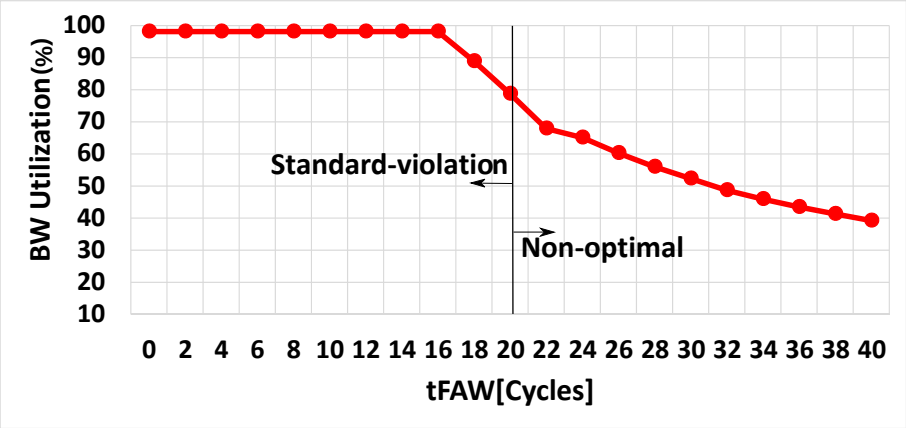


Figure 6.21: $tFAW$ validation results.

Diagnosis. Figure 6.21 delineates the observed utilization for different $tFAW$ values. Lemma 6.9 calculates the golden utilization we expect for the ideal $tFAW$ value.

Lemma 6.9. *Executing $Test_{FAW}$, the BW utilization of the DRAM under test can be calculated as: $\frac{8 \cdot tBUS}{2 \cdot tFAW}$.*

Proof. Executing $Test_{FAW}$, the DRAM under test repeats the behaviour shown in Figure 6.6 (the same figure that illustrates the command sequence for the XOR address mapping) every 8 requests. Focusing on one repetition, the data bus is busy for $8 \cdot tBUS$ cycles. In addition, the 8 requests encounter a total DRAM access latency of $\text{MAX}(2 \cdot tFAW, 8 \cdot tRRD) = 2 \cdot tFAW$. □

6.6.4 tRTRS

Test plan. $Test_{RTRS}$ formalizes the plan to validate $tRTRS$. It is a stream of n read accesses targeting the same row where each access is targeting a different rank than the previous access.

Test 6.8: $Test_{RTRS}$ to validate $tRTRS$.
1 $Test_{RTRS} = [Req_1, Req_2, ..., Req_n]$ 2 where $Req_k = \langle Addr_k, R \rangle, k \in [1, n]$ 3 and $(rnk_l \neq rnk_m) \wedge (rw_l = rw_m)$, $\forall l \; \forall m \in [1, n]$.

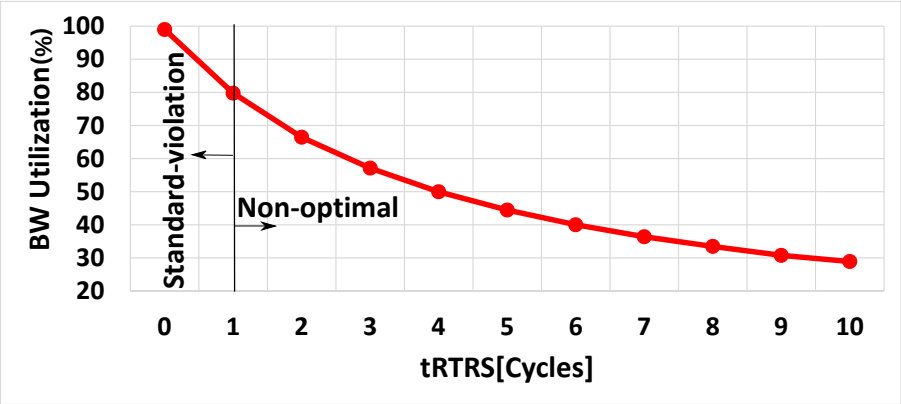


Figure 6.22: $tRTRS$ validation results.

Specifications. The LTL in Spec.6.9 specifies a state in the FSM, where the number of requests is 10 and the $tRTRS$ constraint is exercised 9 times on the explored path to reach this state. This ensures that every two consecutive requests in the test target different ranks.

LTL Specification 6.9: $tRTRS$.

```
G!((num_requests=10)&(num_RANK_TO_RANK_DELAY=9)&
((num_tRL=1)|(num_tWL=1)) & (num_tBUS=1))
```

Diagnosis. Figure 6.22 depicts the observed utilization for different values. Lemma 6.10 calculates the golden utilization we expect for the ideal $tRTRS$ value.

Lemma 6.10. *Executing $Test_{RTRS}$, the BW utilization of the DRAM under test can be approximated to: $\frac{tBUS}{tBUS+tRTRS}$.*

Proof. Since $Test_{RTRS}$ has n requests, the data bus is busy for $n \cdot tBUS$ cycles. Using Figure 6.23, the total DRAM access time can be calculated as: $tRCD+(n-1)\cdot(tBUS+tRTRS)+tRL+tBUS$. As a result, the BW utilization of the DRAM under test is:

$$\frac{n \cdot tBUS}{tRCD + (n-1) \cdot (tBUS + tRTRS) + tRL + tBUS}$$

If $n >> 10$, the BW utilization can be approximated to $\frac{tBUS}{tBUS+tRTRS}$. □

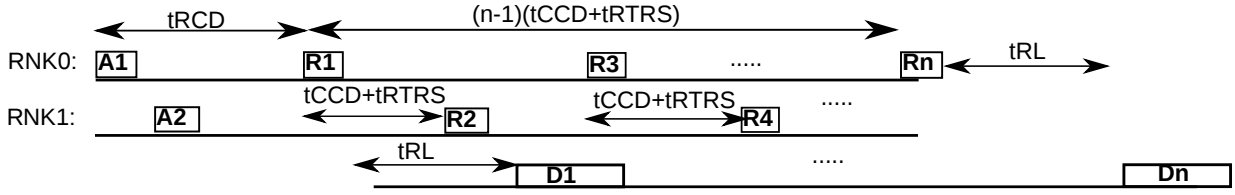


Figure 6.23: Command sequence of $Test_{RTRS}$.

6.6.5 tRTP

Test plan. Studying the state graph in Figure 6.3, a valid command sequence encompassing $tRTP$ would be an A command followed by one or more R commands then a P command to close the row followed by an A to a different row. However, this sequence includes the $tRCD$, $tCCD$, and tRP constraints as well. Therefore, as the dependency graph in Figure 6.16 illustrates, these constraints need to be validated before $tRTP$. Since we already validated them (tRP is included in tRC), we are going to validate $tRTP$. In addition, the number of R commands must be large enough to dominate the $tRAS$ constraint between A and P.

Specifications. Spec. 6.10 shows the LTL property to validate $tRTP$, where `num_tRTP` is the number of occurrences of the $tRTP$ constraint. As Figure 6.24 illustrates, there are two paths of constraints between the A and the P commands. The first is $tRAS$ and the second consists of $tRCD$, a number of $tCCD$ constraints that depends on the number of R requests, and finally $RtoP$. Spec. 6.10 ensures that the second path dominates the first one to show the effect of $tRTP$ constraint on the utilization.

LTL Specification 6.10: $tRTP$.

```
G!(((value_READ_TO_PRE_DELAY*num_READ_TO_PRE_DELAY
 +num_tCCD*value_tCCD+value_tRCD)>(value_tRAS)) &
  (num_READ_TO_PRE_DELAY>0) & ((num_Rx_d=0) & (num_Rd_s=0))
```

Diagnosis. To validate $tRTP$, we compare the observed utilization (Uti_o) from executing $Test_{RTP}$ with the calculated utilization (Uti_c) from Lemma 6.11. Based on the comparison, we make the conclusions tabulated in Table 6.2.

Lemma 6.11. *Executing $Test_{RTP}$, the BW utilization of the MC under test is:*

$$\frac{4tBUS}{tRCD + 3tCCD + tBUS + tRTP + tRP}$$

.

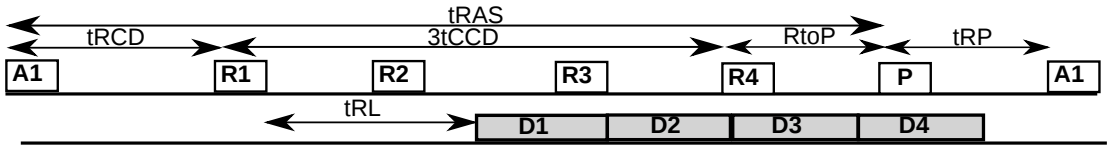


Figure 6.24: Command sequence of $Test_{RTP}$.

Table 6.2: Validating $tRTP$.

$Uti_o = Uti_c$	optimal value	Figure 6.25 at $tRTP = 5$
$Uti_o > Uti_c$	violated	Figure 6.25 at $tRTP < 5$
$Uti_o < Uti_c$	non-optimal value	Figure 6.25 at $tRTP > 5$

Proof. Executing $Test_{RTP}$, the MC under test repeats the behaviour shown in Figure 6.24 every 4 requests. Focusing on one repetition, the data bus is busy for $4 \cdot tBUS$ cycles. In addition, the 4 requests encounter a total DRAM access latency of $tRCD+3tCCD+tBUS+tRTP+tRP$. □

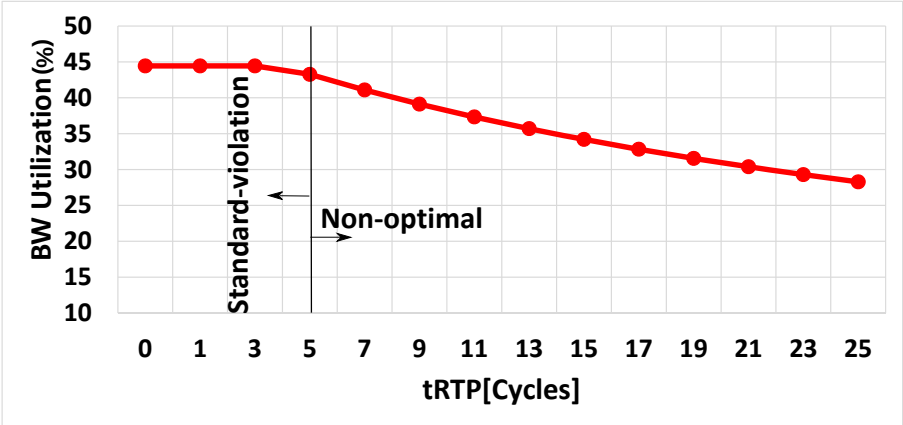


Figure 6.25: $tRTP$ validation results.

6.6.6 tRCD,tWL, and tRL

Test plan. The target is to validate tRL and tWL parameters, which requires two tests. One request is a read operation and the other is a write operation.
Specifications. The LTL in Spec.6.11 specifies a state in the FSM, where the explored path

encounters an A followed by a R. Similarly, the LTL in Spec. 6.12 describes a state in the FSM, where the explored path encounters an A followed by a W. Since a read or a write from a DRAM row requires first to activate that row, it is not possible to exclude the $tRCD$ parameter (between A and R or W). As a consequence, we validate $tRCD, tRL$ and tWL together.

LTL Specification 6.11: $tRCD\_tRL$.

```
G!((num_requests=1) & (num_tRL=1) & (num_tBUS=1))
```

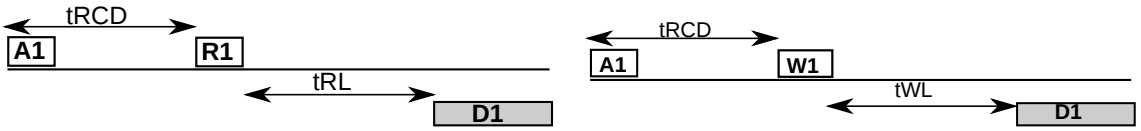
LTL Specification 6.12: $tRCD\_tWL$.

```
G!((num_requests=1) & (num_tWL=1) & (num_tBUS=1))
```

Diagnosis. To validate the parameters $tRCD, tRL$ and tWL, we investigate the utilization observed (Uti_o) from running tests $Test_{\{tRCD,tRL\}}$ and $Test_{\{tRCD,tWL\}}$. If the observed utilization coincides with the calculated utilization (Uti_c) in Lemma 6.12 for both tests, then all the three parameters are set to the standard value. For the DDR3 module used in our validation, this situation is observed in Figures 6.27a, 6.27b and 6.27c at $tRCD = 10$, $tRL = 10$ and $tWL = 9$. Table 6.3 summarizes our debugging conclusions from the utilization graphs. We assume a single parameter is possibly violated at a time.

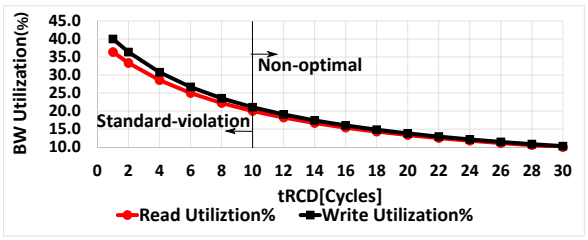
Lemma 6.12. *Executing $Test_{\{tRCD,tRL\}}$, the utilization of the MC under test is: $\frac{tBUS}{tRCD+tRL+tBUS}$. similarly, executing $Test_{\{tRCD,tWL\}}$, the BW utilization of the MC under test is: $\frac{tBUS}{tRCD+tWL+tBUS}$.*

Proof. The proof can be easily derived from the behaviour in Figures 6.26a and 6.26b for the read and write requests, respectively. □

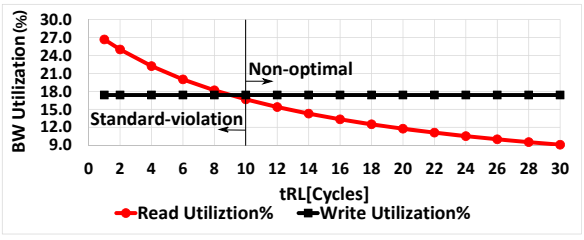


(a) $Test_{tRCD,tRL}$. (b) $Test_{tRCD,tWL}$.

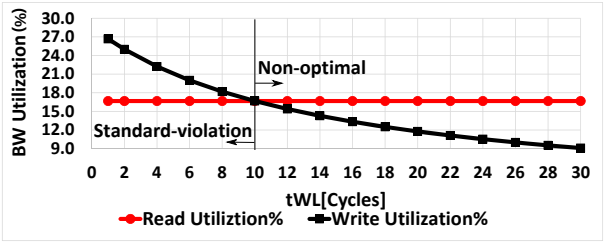
Figure 6.26: Validating $tRCD$, tRL, and tWL.



(a) tRCD



(b) tRL



(c) tWL

Figure 6.27: $tRCD$, tRL, and tWL validation results.

Table 6.3: Validating $tRCD$, tRL and tWL.

$Test_{\{tRCD,tRL\}}$	$Test_{\{tRCD,tWL\}}$	Conclusion	Figure
$Uti_o > Uti_c$ $Uti_o < Uti_c$	$Uti_o > Uti_c$ $Uti_o < Uti_c$	$tRCD$ is violated. $tRCD$ is not optimal.	6.27a
$Uti_o > Uti_c$ $Uti_o < Uti_c$	$Uti_o = Uti_c$ $Uti_o = Uti_c$	tRL is violated. tRL is not optimal.	6.27b
$Uti_o = Uti_c$ $Uti_o = Uti_c$	$Uti_o > Uti_c$ $Uti_o < Uti_c$	tWL is violated. tWL is not optimal.	6.27c

6.6.7 tRRD

Test plan. $Test_{RRD}$ formalizes the test plan for validating $tRRD$. It is a sequence of four read accesses targeting the same rank where each access is targeting a different bank than the previous access.

Specifications. The LTL in Spec. 6.13 describes a state, where the path between the initial state and this state exercises the $tRRD$ constraint 3 times. In addition, this path encountered three R

Test 6.9: $Test_{RRD}$ to validate $tRRD$.

1 $Test_{RRD} = [Req_1, Req_2, Req_3, Req_4]$
2 where $Req_k = \langle Addr_k, R \rangle, k \in [1, 4]$
3 and $(rnk_l = rnk_m) \wedge (bnk_l \neq bnk_m), \forall l \; \forall m \in [1, 4]$.

commands to a different bank in the same bank than the first R command.

LTL Specification 6.13: $tRRD$.

```
G! ((num_tRRD=3)&(num_Rd_s=3)&(num_tRL=1))
```

Diagnosis. Figure 6.28 illustrates the observed utilization for different $tRRD$ values. Lemma 6.13 determines the golden utilization expected for the ideal $tRRD$ value.

Lemma 6.13. *Executing $Test_{RRD}$, the BW utilization of the DRAM under test can be calculated as:* $\frac{4 \cdot tBUS}{3 \cdot tRRD + tRCD + tRL + tBUS}$.

Proof. Executing $Test_{RRD}$, the DRAM under test shows the behaviour shown in Figure 6.29 for the 4 requests. The data bus is busy for $4 \cdot tBUS$ cycles. In addition, the 4 requests encounter a total DRAM access latency of $3 \cdot tRRD + tRCD + tRL + tBUS$. □

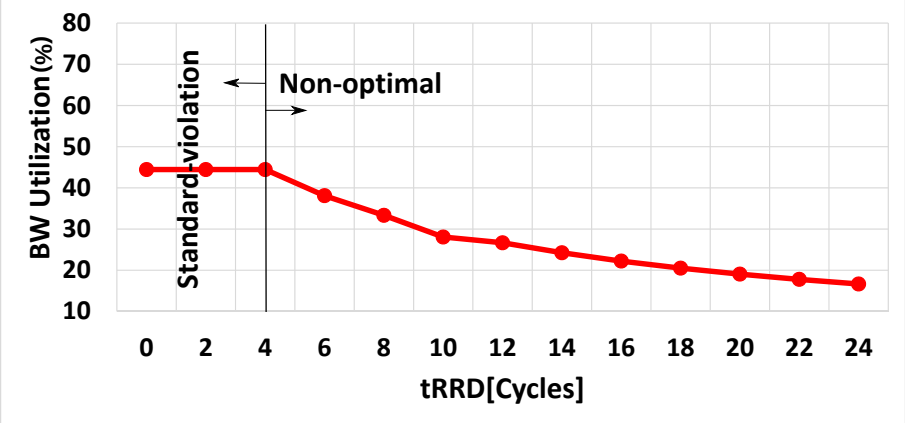


Figure 6.28: $tRRD$ validation results.

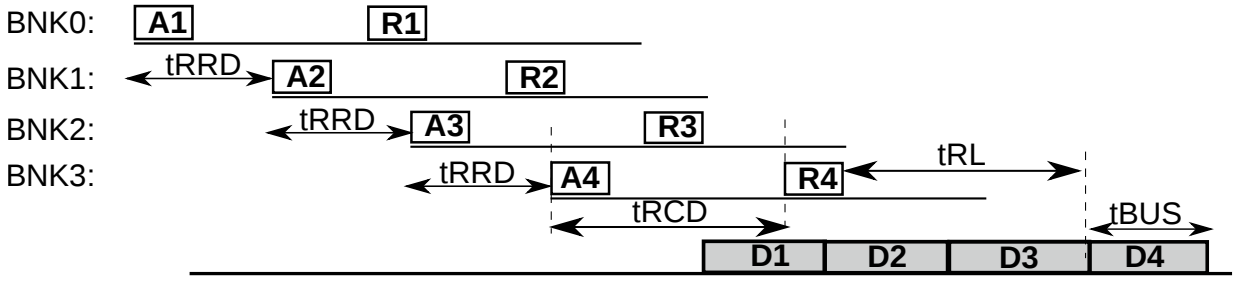


Figure 6.29: Command sequence of $Test_{RRD}$.

6.6.8 tWR

Test plan. $Test_{WR}$ formalizes the plan to validate tWR. It is a stream of n write accesses targeting the same bank, where each access is targeting a different row than the previous access (0% row locality).

Test 6.10: $Test_{WR}$ to validate tWR.

1 $Test_{WR} = [Req_1, Req_2, ..., Req_n]$
2 where $Req_k = \langle Addr_k, W \rangle, k \in [1, n]$
3 and $(bnk_l = bnk_m) \wedge (rw_l \neq rw_m), \forall l \, \forall m \in [1, n]$.

Specifications. The LTL in Spec. 6.14 encodes the test plan to discover any violations in the tWR constraint. It specifies a state in the FSM where there are 10 requests. The property `(num_Wx_d=0) & (num_Wd_s=0)` ensures that all requests target the same bank. the property `(num_WRITE_TO_PRE_DELAY=9)` dictates that, except the first request, each request has to pre-charge the open row in the row buffer before it activates its own row.

LTL Specification 6.14: tWR.

```
G!((num_tRCD = 10) & (num_W_to_P_DELAY=9) &
   (num_Wx_d=0) & (num_Wd_s=0) )
```

Diagnosis. Figure 6.30 shows the observed utilization for different tWR values. Lemma 6.14 calculates the golden utilization we for the ideal tWR value.

Lemma 6.14. *Executing $Test_{WR}$, the BW utilization of the DRAM under test is:* $\frac{tBUS}{tRCD+tWL+tBUS+tWR+tRP}$.

Proof. Executing $Test_{WR}$, each DRAM access is translated into the commands shown in Figure 6.31. Therefore, Each request is serviced in $tRCD + tWL + tBUS + tWR + tRP$ cycles while it utilizes the data bus for only $tBUS$ cycles. □

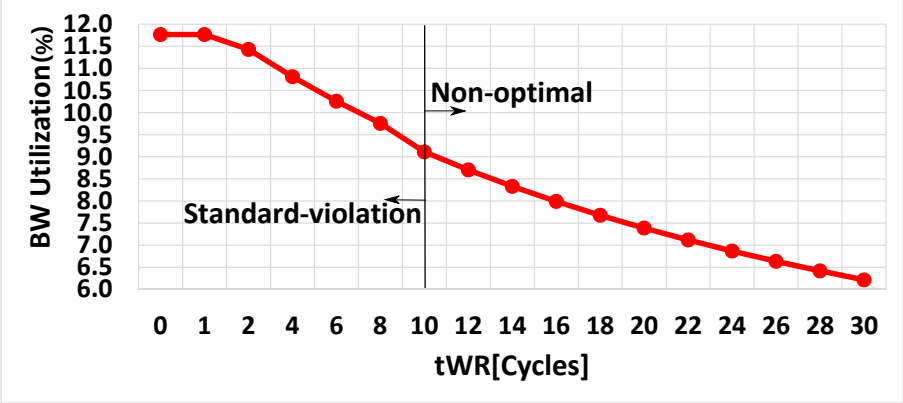


Figure 6.30: tWR validation results.

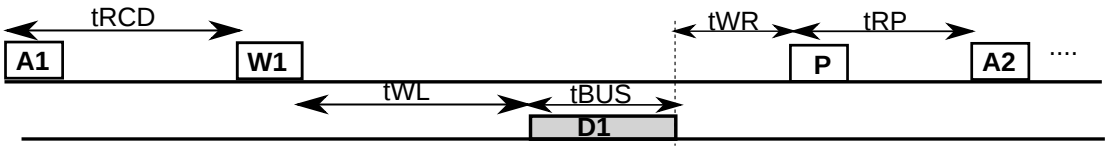


Figure 6.31: Command sequence of $Test_{WR}$.

6.6.9 tWTR

Test plan. $Test_{WTR}$ formally describes the plan to validate $tWTR$. It is a stream of n accesses targeting the same bank and row where each access is of different type than the previous access (100% R/W switching).

Algorithm 6.1: $Test_{WTR}$: Test to validate $tWTR$.
1 $Test_{WTR} = [Req_1, Req_2, ..., Req_n]$
2 where $Req_k = \langle Addr_k, Type_k \rangle, k \in [1, n] and nisodd$
3 and $(bnk_l = bnk_m) \wedge (rw_l = rw_m) \wedge (ty_l \neq Type_{l-1}), \forall l \, \forall m \in [1, n], n \gg 10$.

Specifications. Spec. 6.15 shows the LTL for validating $tWTR$. Since each two successive requests are of different type, 5 requests out of 10 suffers from a $WtoR_B$ delay. As a result, `(num_W_to_R_DELAY_B=5)`.

LTL Specification 6.15: $tWTR$.

```
G!((num_requests = 10)&(num_tRCD=1)&
    (num_W_to_R_DELAY_B=5)&(num_Rx_d=0)&(num_Rd_s=0))
```

Diagnosis. Figure 6.32 delineates the observed utilization for different $tWTR$ values. Lemma 6.15 calculates the golden utilization we for the ideal tWR value.

Lemma 6.15. *Executing $Test_{WTR}$, the BW utilization of the DRAM under test can be approximated to: $\frac{tBUS}{tRL+tWTR+tBUS+tRTRS}$.*

Proof. Since $Test_{WTR}$ has n requests, the data bus is busy for $n \cdot tBUS$ cycles. Using Figure 6.33, the total DRAM access time can be calculated as:

$$tRCD + \frac{n-1}{2} \cdot (tRL + tWTR + 2tBUS + tRTRS) + tCL + tBUS.$$

Where:

$$tCL = \begin{cases} tRL, & \text{if } Type_n = R \\ tWL, & \text{if } Type_n = W \end{cases}$$

As a result, the BW utilization of the DRAM under test is:

$$\frac{n \cdot tBUS}{tRCD + \frac{n-1}{2} \cdot (tRL + tWTR + 2tBUS + tRTRS) + tCL + tBUS}.$$

If $n >> 10$, the BW utilization can be approximated to

$$\frac{2tBUS}{tRL + tWTR + 2tBUS + tRTRS}.$$

□

6.6.10 Summary

In summary, to validate a timing parameter we conduct the following procedure. 1) We execute the corresponding test from Table 6.4. 2) We compare the observed utilization with the calculated utilization. 3) Based on the comparison, we determine whether the parameter under test is: 3.a) compliant with the standard, 3.b) violated, or 3.c) set to a non-optimal value. We tabulate calculated utilizations from all tests in Table 6.4. In Table 6.4, unless specified, the number of requests is $n \gg 10$.

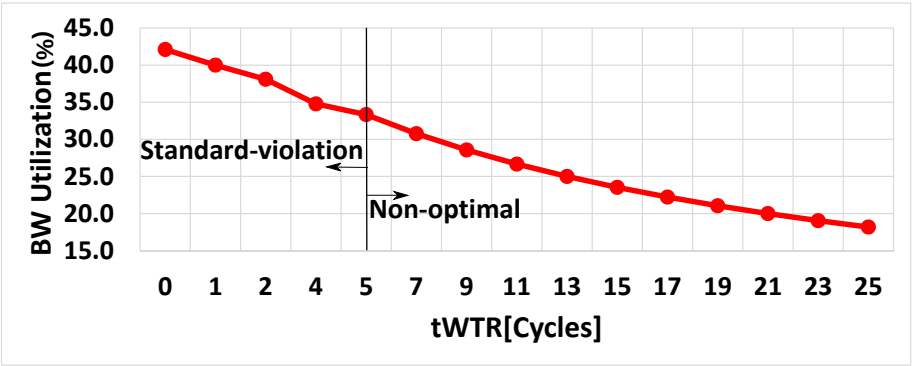


Figure 6.32: $tWTR$ validation results.

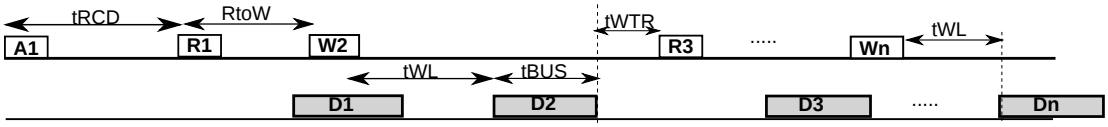


Figure 6.33: Command sequence of $Test_{WTR}$.

Test	Conditions ($\forall l \in [1, n]$)	Utilization
tRC	$(bnk_l = bnk_{l-1}) \wedge (rw_l \neq rw_{l-1})$	$\frac{tBUS}{tRC}$
$tCCD$	$(bnk_l = bnk_{l-1}) \wedge (rw_l = rw_{l-1})$	$\frac{tBUS}{tCCD}$
$tFAW$	$(rnk_l = rnk_{l-1}) \wedge$ $(bnk_l \neq bnk_{l-1}) \wedge (rw_l \neq rw_{l-1})$	$\frac{8 \cdot tBUS}{2 \cdot tFAW}$
$tRTRS$	$(rnk_l \neq rnk_{l-1}) \wedge (rw_l = rw_{l-1})$	$\frac{tBUS}{tBUS+tRTRS}$
$tRRD$	$(n = 4) \wedge$ $(rnk_l = rnk_{l-1}) \wedge (bnk_l \neq bnk_{l-1})$	$\frac{4 \cdot tBUS}{3 \cdot tRRD+tRCD+tRL+tBUS}$
tWR	$(ty_l = W) \wedge$ $(bnk_l = bnk_{l-1}) \wedge (rw_l \neq rw_{l-1})$	$\frac{tBUS}{tRCD+tWL+tBUS+tWR+tRP}$
$tWTR$	$(ty_l \neq ty_{l-1}) \wedge$ $(bnk_l = bnk_{l-1}) \wedge (rw_l = rw_{l-1})$	$\frac{tBUS}{tRL+tWTR+tBUS+tRTRS}$

Table 6.4: Tests of timing parameters.

6.6.11 Smart Refresh

In this section, we validate the behaviour of the *Smart Refresh* technique [153]. This serves three purposes: 1) validating a feature involving the REF command, 2) validating one of the DRAM power reduction techniques, and 3) illustrating the usage of a different metric than the bandwidth utilization with MCXplore. The main observation behind Smart Refresh is that a normal access to a DRAM row plays the same role as a refresh command from the data restoration standpoint. Accordingly, Smart Refresh aims to reduce power consumption by skipping refreshing recently accessed rows. The memory controller maintains a per-row counter and issues a refresh command to a row when its counter value reaches zero. Upon accessing a row, its corresponding counter is set to its maximum value. We implement Smart Refresh in DRAMsim2, where the controller maintains a 2-bit counter per row.

Test Generation. The number of REF commands issued by Smart Refresh is supposed to decrease upon increasing the number of accessed rows in the tests. Therefore, we use MCXplore to generate tests with different number of A commands as Specification 6.16 describes. To capture a considerable number of refreshes, each test has one million requests, and we run the simulation for 10 million cycles.

LTL Specification 6.16: Smart Refresh validation (nACT varies from 8 to 1000000 in a step of 8).

```
G!((num_requests = 1000000)&(num_ACT=nACT)
```

Diagnosis. We use the total aggregated numbers of issued A and REF commands as our metrics in this experiment. These numbers can be obtained using performance counters such as those existing in Intel processors [154]. We depict the results for the Smart Refresh technique along with the baseline refresh mechanism in Figure 6.34. The baseline issues a fixed number of REF commands agnostically to the access pattern. On the other side, Smart Refresh, as expected, issues less number of REF commands for tests accessing more rows. Figure 6.34 shows that for tests with small number of accessed rows, Smart Refresh acts exactly as the baseline refresh mechanism (until point ❶), which validates the worst scenario. Contrarily, for tests accessing large number of rows (after point ❷), Smart Refresh does not need to issue any extra REF command. This aligns with the ideal scenario represented in [153].

6.6.12 Command Bus Contention

All banks of a DRAM rank share the command and data bus. Accordingly, if more than one DRAM command are ready at a specific cycle, the MC must have a policy to select only one command to issue to prevent bus collisions. In this context, a ready command is the command

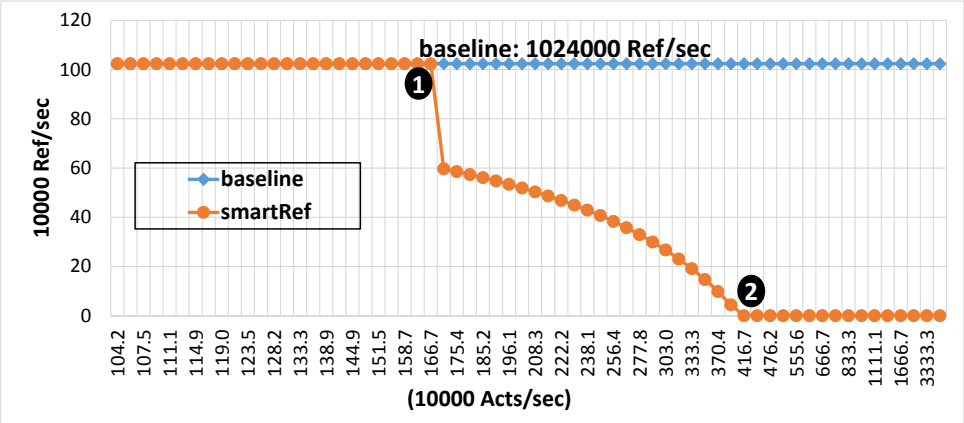
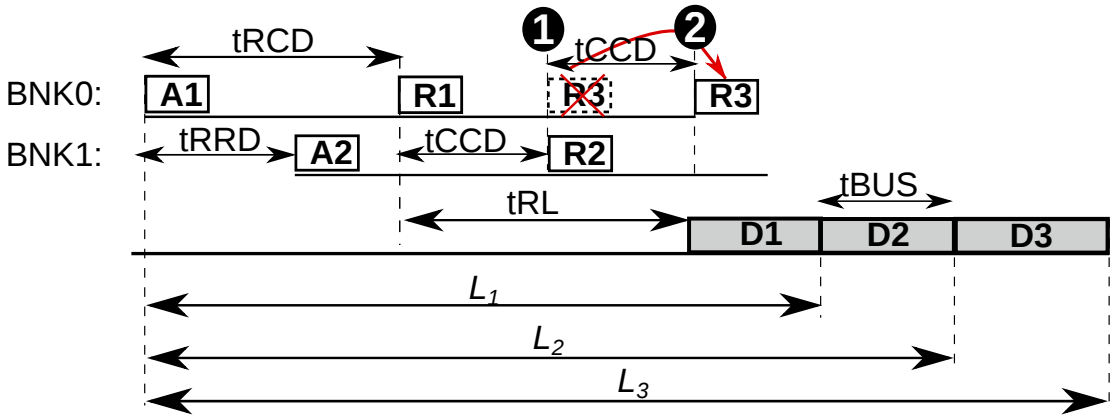


Figure 6.34: Smart Refresh behaviour with different number of accessed rows.

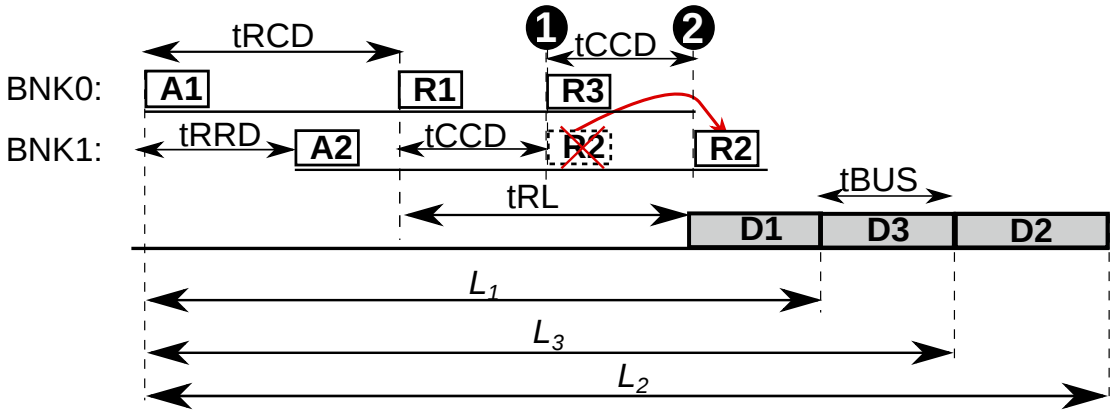
that satisfies all the corresponding timing constraints and can be issued according to the command arbitration policy. For example, commands R2 and R3 in Figure 6.35 are ready on the same cycle; however, only one of them can be issued at one cycle. One possible policy is to favor ready requests to different banks to increase DRAM parallelism through interleaving (Figure 6.35a). An alternative policy is to favor ready requests to same bank to increase DRAM locality through row hits (Figure 6.35b).

Test Generation. The target is to generate a test pattern that exhibit a bus contention, i.e. having more than one ready command to be issued on the same cycle. We generate a test with three memory requests, Req_1, Req_2, and Req_3 . Req_1 and req_3 target the same row in the same bank, while Req_2 targets a different bank. As Figure 6.35 illustrates, both R2 and R3 are ready on the same cycle. Accordingly, one of the commands is issued on cycle ❶, while the other has to be delayed $tCCD$ cycles to be issued on cycle ❷.

Diagnosis. There are two possibilities for the test pattern, either R2 is postponed or R3. To verify the policy resolving the command bus contention, we use the request latency as our metric since Req_2 and Req_3 will have different latencies based on the implemented policy. L_i in Figure 6.35 is the latency of Req_i. Bandwidth utilization cannot be used in this case since both possibilities lead to the same utilization. Based on the monitored latency, we can figure out which R command is postponed by the MC and compare this to the expected behavior. For instance, in Figure 6.35a, the MC postpones R3; thus, $L_3 > L_2$. On the other hand, in Figure 6.35b, the MC postpones R2 such that $L_3 < L_2$.



(a) Prioritizing accesses to different banks (favor bank interleaving).



(b) Prioritizing hit accesses to same bank (favor row hits).

Figure 6.35: Policies to resolve command bus contention.

6.7 Extensibility of MCXplore

MCXplore treats extensibility as a first-class citizen. If a designer wants to validate a new policy, she only needs to identify the properties of that policy and encode them in LTL specifications. Afterwards, she chooses the suitable model based on whether the policy is at the frontend or the backend, and MCXplore will generate the desired test suites. Furthermore, leveraging the modularity of MCXplore, a validation engineer can easily integrate any other model than the provided two models. She only needs to modify the interface functions that parse the model to generate the required test. We encourage researchers and validation engineers to extend and use MCXplore to validate and test their proposed MC's policies.

6.8 Summary

In this chapter, we proposed a framework for validating MC designs. We introduced two models for the test input of the MC enabling validation engineers and researchers to specify their test plan as specifications in temporal logic. We used model checking to generate test templates that satisfy this plan. We implemented this framework and released it open-source as MCXplore, accompanied with a regression test suite for validating basic MC features. Using MCXplore, we showed how to validate the correctness of state-of-the-art MC features as well as discover timing violations in the DRAM subsystem.

Chapter 7

Bounding Total Memory Latency in Multi-Core Real-Time Systems

In Chapter 2, we calculated the WCL incurred by any request upon accessing the shared memory bus connecting cores to the shared cache. In chapter 3, we derived the WCL incurred by any request due to the coherence interference amongst cores. In chapter 4, we bounded the WCL incurred by any request upon accessing the DRAM. All latencies derived in these chapters are per request. In this chapter, we integrate the outcomes of these chapters to derive the total memory latency bound incurred by the task. Recall that the total WCET of a task is the summation of the WC computation time and the WC memory latency of this task. Obtaining the total memory latency at the task level is necessary to be able to compute this WCET. Conducting the analysis at the task level, as opposed to the per-request analysis, enables considering different scenarios of memory requests (such as cache hits vs. cache misses), which is a key to derive more realistic tighter WCETs. Recall from Chapter 3, the analysis for PMSI assumes a TDM bus arbitration, we first compute the coherence interference assuming the bus is managed using CArb.

7.1 System Model

Figure 7.1 illustrates the system model considered in this chapter.

- We consider a system with N requestors, a private per-core L1 cache, a shared on-chip L2 cache, and a shared off-chip DRAM amongst requestors.
- Each task is assigned a core, while tasks do not share cores.

- The system deploys the CArb arbiter proposed in Chapter 2 to schedule accesses to the shared L2 cache. The slot width of the CArb schedule is S^{L2}. Similar to Chapter 3, S^{L2} is chosen to account for one memory transfer between the core and the L2 in addition to the overhead resulting from necessary coherence messages.
- At the end of each hyperperiod of CArb's schedule, one additional slot is granted to each task. This slot is dedicated to conduct write back operations as a response to other tasks requesting a cache line that is modified by this task. Figure 7.1 illustrates this modification to the CArb's schedule. Recall that the total number of classes is n, while the total number of tasks in each class l is Γ_l. Accordingly, the WC arbitration latency calculated in Lemma 2.1, can be recalculated as:

$$WCL_{jl}^{arb} = \left(\left(\sum_{v=1, v\neq j}^{v=\Gamma_l} \left\lceil \frac{\tau W_{vl}}{\tau W_{jl}} \right\rceil \right) + \left(\left\lceil \frac{\tau P_{jl}}{Z_l} \right\rceil \times \sum_{\substack{\forall e | (e \neq l \wedge \\ f_e \in \chi_l)}} \left(\left\lceil \frac{CW_e}{CW_l} \right\rceil \times Z_e \right) \right) + \left(\sum_{l=1}^{n} \Gamma_l \right) + 1 \right) \times S^{L2} \tag{7.1}$$

- Data coherence is maintained using the PMSI coherence protocol proposed in Chapter 3.
- Accesses to the DRAM are managed using the PMC memory controller proposed in Chapter 4.

7.2 Timing Analysis of Coherence Interference Assuming CArb's Schedule

Lemma 7.1. *The WC coherence latency of task τ_{jl} in a system deploying CArb's arbitration is calculated as follows:*

$$WCL_{jl}^{coh} = \left(\sum_{\forall e} \sum_{\forall v} WCL_{ve}^{arb} \right) + WCL_{jl}^{arb} \quad \textit{where:} \quad \{e \in [1,n], v \in [1, \Gamma_e] | \quad e \neq l || v \neq j\}$$

Proof. From Lemma 3.2, the WC occurs when the task under analysis, τ_{jl}, requests a cache line that is currently modified or requested to modify by all other tasks. Accordingly, τ_{jl} has to wait

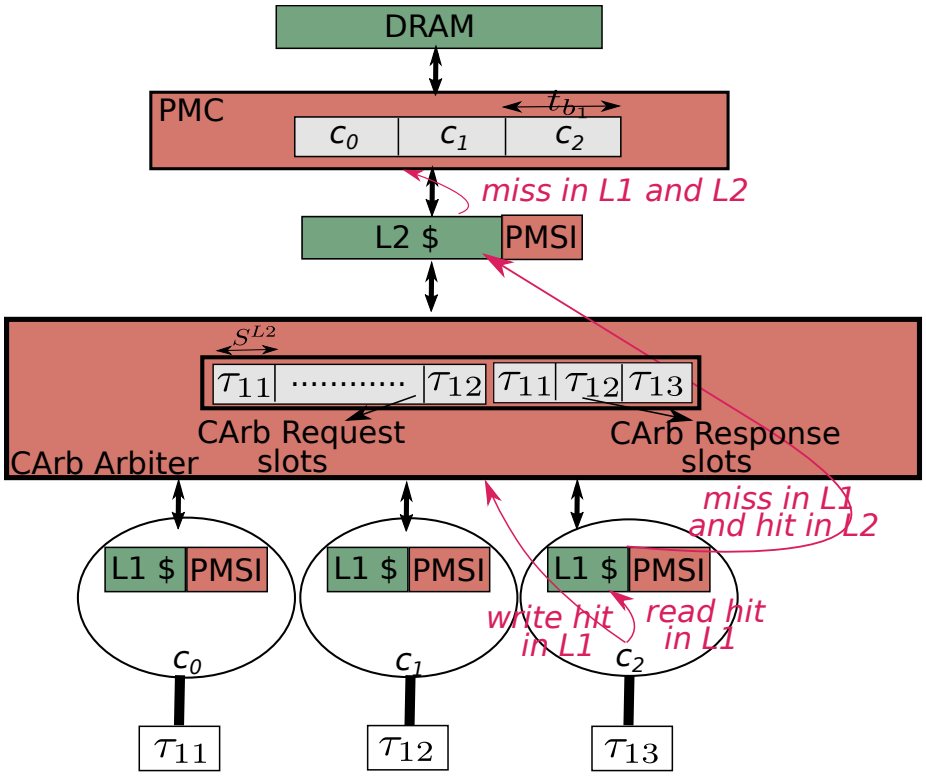


Figure 7.1: An architectural example for the system model used in deriving the total WC memory aggregated latency. The system has three cores and three tasks, where τ_{11} is mapped to c_0, τ_{12} is mapped to c_1, and τ_{13} is mapped to c_2.

for each other task to obtain, modify, and write back that line. Each task needs two slots to perform these operations: a request slot and a response slot. Since each task τ_{ve} gets these two slots every WCL_{ve}^{arb} cycles, τ_{jl} has to wait for a total of $\sum_{\forall e}\sum_{\forall v} WCL_{ve}^{arb}$ before the requested cache line can be sent to it. Moreover, similar to the proof of Lemma 3.2, when the shared memory has the updated version that is ready to send to τ_{jl}, τ_{jl} might have just missed its slot in the current period. As a consequence, it has to wait for additional WCL_{jl}^{arb} cycles. Finally, since Equation 7.1 accounts for the latency suffered due to the write back responses (i.e. intra-core coherence latency), the total coherence latency can be calculated as: $\left(\sum_{\forall e}\sum_{\forall v} WCL_{ve}^{arb}\right) + WCL_{jl}^{arb}$. □

7.3 Aggregated Memory Latency Analysis

Lemma 7.2. *The total aggregated WC memory latency of a task τ_{jl}, MAL_{jl}, can be computed as:*

$$MAL_{jl} = L1_{jl}^{Rhits} \times L1^{accL} + L1_{jl}^{Whits} \times (WCL_{jl}^{arb}) + L2_{jl}^{hits} \times (WCL_{jl}^{arb} + WCL_{jl}^{coh}) + L2_{jl}^{misses} \times WCL_{jl}^{DRAM}.$$

Where:

- $L1_{jl}^{Rhits}$ *is the number of read requests issued by* τ_{jl} *and are hits in L1 cache.*
- $L1_{jl}^{Whits}$ *is the number of write requests issued by* τ_{jl} *and are hits in L1 cache.*
- $L2_{jl}^{hits}$ *is the number of requests issued by* τ_{jl} *and are hits in L2 cache.*
- $L2_{jl}^{misses}$ *is the number of requests issued by* τ_{jl} *and are misses in L2 cache.*
- $L1^{accL}$ *is the WCL of a read request issued by* τ_{jl} *upon accessing L1, which is the hit latency of the L1 cache.*
- WCL_{jl}^{arb} *is the WCL of a write request issued by* τ_{jl} *upon accessing L1, which is calculated by Equation 7.1.*
- WCL_{jl}^{coh} *is the WCL of a read or write request issued by* τ_{jl} *upon accessing L2, which is defined by Lemma 7.1.*
- WCL_{jl}^{DRAM} *is the WCL of a read or write request issued by* τ_{jl} *upon accessing DRAM, which is calculated in Equation 4.7.*

Proof. As Figure 7.1 illustrates, each memory request by τ_{jl} incurs one of four possible situations: 1) it is a read hit in the L1 private cache, 2) it is a write hit in L1 private cache, 3) it is a miss in L1 but a hit in the L2 shared cache, or 3) it is a miss in L1 and L2, thus has to access the DRAM.

We consider each memory level separately as follows:

1. **Read hit in L1 private cache.** Since L1 cache is private for each core (and hence, for the task running on that core), accesses to L1 do not encounter interference from other tasks' accesses. Further, since the request is a read, it does not need to wait for the core slot in the bus. Accordingly, the access latency to L1 can be considered constant and equals to the L1

hit latency specified by the processor datasheet ($WCL^{L1} = L1^{accL}$). If task τ_{jl} has a total of $L1_{jl}^{Rhits}$ requests that are read hits in its L1 private cache, the total WCL of these requests is $L1_{jl}^{Rhits} \times L1^{accL}$.

2. **Write hit in L1 private cache.** Since a write hit request needs to issue coherence messages on the bus, it has to wait for the designated task slot before it can proceed. Accordingly, in WC, a write hit has to wait for the WC arbitration latency, WCL_{jl}^{arb} which is calculated by Equation 7.1. If τ_{jl} has a total of $L1_{jl}^{Whits}$ requests that are write hits in its L1 private cache, the total WCL of these requests is $L1_{jl}^{Whits} \times WCL_{jl}^{arb}$.

3. **L2 shared cache.** The WCL of any request that is a hit in the shared L2 cache in a system deploying CArb arbitration and PMSI can be computed as: $WCL_{jl}^{arb} + WCL_{jl}^{coh}$. Since there are $L2_{jl}^{hits}$ requests that are hits in L2 from τ_{jl}, the total WC latency that τ_{jl} suffers at L2 is: $L2_{jl}^{hits} \times (WCL_{jl}^{arb} + WCL_{jl}^{coh})$.

4. **Shared DRAM.** Since the DRAM accesses are manged using PMC, the WCL_{jl}^{DRAM} is the PMC's upper bound latency defined in Equation 4.7. Assuming that requestor i is used to execute task τ_{jl}, the WC DRAM latency is $WCL_{jl}^{DRAM} = UBL_i$.

From 1–4, the MAL_{jl} can be calculated as Lemma 7.2 depicts. □

Chapter 8

Conclusion and Future Work

Using multi-core platforms to implement real-time systems is promising because of their low-cost and high-performance benefits. Nonetheless, interference amongst cores competing to access shared memory resources is a challenge facing the deployment of real-time systems in multi-core platforms. This thesis presents techniques to bound this interference and provides predictable shared memory resources. This includes:

1. A predictable shared bus arbitration technique that is aware of each task's timing requirements and criticality.

2. A predictable cache coherence protocol that allows co-running tasks to predictably and simultaneously share data.

3. A predictable DRAM memory controller that deploys a mixed-page policy and optimal harmonic distributed TDM arbitration to grant differential services to different tasks based on their latency and bandwidth requirements.

4. A latency-based analysis that produces best- and worst-case memory latency bounds for memory accesses to the DRAM. We use this analysis to reverse-engineer architectural details of embedded memory controllers. Knowing these details helps in developing compiler and source-code optimizations to achieve predictability in COTS platforms.

5. A timing analysis that integrates the proposed approaches at different levels of the memory hierarchy to provide bounds for the total aggregated memory latency incurred by a task.

Techniques proposed in this thesis enables multiple future extensions. This includes:

1. A full support for tasks with mixed criticalities at all levels of the memory hierarchy. Currently, the shared bus and the memory controller are both criticality-aware, while the cache coherence protocol does not support tasks with different criticalities. Since emerging domains such as automotive industry adopt tasks with different criticalities, it is important to develop criticality-aware approaches that span the whole memory hierarchy.

2. Developing a framework that integrates all these approaches to provide the real-time community with a fully predictable memory system that can be incorporated with core scheduling techniques or timing analysis tools. This framework should be modular to allow the integration of other novel policies (e.g. other memory controllers, bus arbiters, or cache mechanisms). This framework can also help in developing techniques that involve different memory levels together. For example, instead of focusing on achieving tight bounds at the DRAM level or the cache level separately, techniques that consider the interaction between both DRAM and LLC cache may be able to provide tighter bounds.

3. Leveraging the exposed memory controller details towards developing software-level techniques to provide predictable memory accessing in COTS memory systems.

Validating the performance of novel memory policies is vital to fairly compare different approaches and precisely assess their contributions. Moreover, verifying the timing and functional correctness of these policies is indispensable towards the adoption of these policies in commercial platforms. This thesis presents an open-source framework to automate the validation and verification process of the DRAM subsystem including the memory controller. Nonetheless, extending this approach to provide easy-to-adopt techniques to verify and validate other components in the memory hierarchy such as the cache coherence protocol is a potential future work.

References

[1] M. Hassan and H. 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, pp. 1–11.

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

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

[4] M. Hassan, H. Patel, and R. Pellizoni, "PMC: A requirement-aware DRAM controller for multi-core mixed criticality systems," *ACM Transactions on Embedded Computing Systems (TECS)*, 2016, to appear.

[5] M. Hassan, A. M. Kaushik, and H. Patel, "Reverse-engineering embedded memory controllers through latency-based analysis," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015, pp. 297–306.

[6] ——, "Exposing implementation details of embedded dram memory controllers through latency-based analysis," *IEEE Transactions on Computers (TC)*, 2016, under review.

[7] M. Hassan and H. Patel, "MCXplore: An automated framework for validating memory controller designs," in *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 1357–1362.

[8] ——, "MCXplore: Automating the validation process of DRAM memory controller designs," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2016, under review.

[9] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *IEEE International Real-time Systems Symposium (RTSS)*, 2007, pp. 239–243.

[10] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, 2011.

[11] JEDEC, "DDR3 SDRAM Standard." [Online]. Available: http://jedec.org

[12] J.-E. Kim, T. Abdelzaher, L. Sha, A. Bar-Noy, R. Hobbs, and W. Dron, "On maximizing quality of information for the internet of things: A real-time scheduling perspective (invited paper)," in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2016, pp. 202–211.

[13] P. Graydon and I. Bate, "Safety assurance driven problem formulation for mixed-criticality scheduling," *International Workshop on Mixed Criticality Systems (WMC), RTSS*, 2013.

[14] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2012, pp. 145–154.

[15] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *IEEE Euromicro Conference onReal-Time Systems (ECRTS)*, 2008, pp. 147–155.

[16] R. M. Pathan, "Schedulability analysis of mixed-criticality systems on multiprocessors," in *IEEE Euromicro Conference on Real-time Systems (ECRTS)*, 2012, pp. 309–320.

[17] H. Li and S. Baruah, "Global mixed-criticality scheduling on multiprocessors," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2012, pp. 166–175.

[18] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, "Mixed-criticality scheduling on multiprocessors," *Real-Time Systems*, vol. 50, no. 1, pp. 142–177, 2014.

[19] I. Bate, A. Burns, and R. I. Davis, "A bailout protocol for mixed criticality systems," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2015, pp. 259–268.

[20] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in *International Conference on Embedded Software (EMSOFT)*, 2013, pp. 1–15.

[21] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *IEEE International Conference on Design, Automation and Test in Europe (DATE)*, 2010, pp. 741–746.

[22] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha, "Coscheduling of cpu and i/o transactions in cots-based embedded systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2008, pp. 221–231.

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

[24] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2013, pp. 157–167.

[25] N. Chetan Kumar, S. Vyas, R. K. Cytron, C. D. Gill, J. Zambreno, and P. H. Jones, "Cache design for mixed criticality real-time systems," in *IEEE International Conference on Computer Design (ICCD)*, 2014, pp. 513–516.

[26] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *IEEE Euromicro Conference on Real-Time System (ECRTS)*, 2012, pp. 299–308.

[27] A. Burns and R. Davis, "Mixed criticality systems: A review," *Department of Computer Science, University of York, Tech. Rep*, 2013.

[28] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 145–154.

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

[30] P. Ekberg and W. Yi, "Schedulability analysis of a graph-based task model for mixed-criticality systems," *Real-Time Systems*, vol. 52, no. 1, pp. 1–37, 2015.

[31] L. Sha, "Real-time virtual machines for avionics software porting and development," in *Real-Time and Embedded Computing Systems and Applications (RTCSA)*. Springer, 2004, pp. 123–135.

[32] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware support for WCET analysis of hard real-time multicore systems," in *ACM SIGARCH Computer Architecture News*, 2009, pp. 57–68.

[33] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, "Weighted round-robin cell multiplexing in a general-purpose ATM switch chip," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 9, no. 8, pp. 1265–1279, 1991.

[34] M.-K. Yoon, J.-E. Kim, and L. Sha, "Optimizing tunable wcet with shared resource allocation and arbitration in hard real-time multicore systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2011, pp. 227–238.

[35] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010, pp. 13–22.

[36] A. Minaeva, P. Sucha, B. Akesson, and Z. Hanzalek, "Scalable and efficient configuration of time-division multiplexed resources," *Elsevier Journal of Systems and Software*, vol. 113, pp. 44–58, 2016.

[37] L. Sigrist, G. Giannopoulou, P. Huang, A. Gomez, and L. Thiele, "Mixed-criticality runtime mechanisms and evaluation on multicores," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015, pp. 194–206.

[38] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-time scheduling using credit-controlled static-priority arbitration," in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008, pp. 3–14.

[39] K. Hyesoon, J. Lee, N. B. Laksminarayana, J. Lin, and T. Pho, "Macsim: A CPU-GPU heterogeneous simulation framework." [Online]. Available: http://code.google.com/p/macsim/

[40] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[41] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *IEEE Dependable Computing Conference (EDCC)*, 2012, pp. 132–143.

[42] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst, "System level performance analysis for real-time automotive multicore and network architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 28, no. 7, pp. 979–992, 2009.

[43] G. Gracioli and A. A. Fröhlich, "On the design and evaluation of a real-time operating system for cache-coherent multicore architectures," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 2, pp. 2–16, 2016.

[44] G. Gracioli, A. Alhammad, and R. Mancuso, "A survey on cache management mechanisms for real-time embedded systems," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, pp. 32:1–32:36, 2015.

[45] D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches," in *IEEE Real-Time Systems Symposium (RTSS)*, 2009, pp. 68–77.

[46] B. Lesage, D. Hardy, and I. Puaut, "Shared data caches conflicts reduction for WCET computation in multi-core architectures." in *IEEE International Conference on Real-Time and Network Systems (RTNS)*, 2010, p. 2283.

[47] J. M. Calandrino and J. H. Anderson, "On the design and implementation of a cache-aware multicore real-time scheduler," in *IEEE Euromicro Conference on Real-Time Systems, (ECRTS)*, 2009, pp. 194–204.

[48] A. Pyka, M. Rohde, and S. Uhrig, "Extended performance analysis of the time predictable on-demand coherent data cache for multi-and many-core systems," in *IEEE Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, 2014, pp. 107–114.

[49] "Predictable cache coherence for multi-core real-time systems." [Online]. Available: https://git.uwaterloo.ca/caesr-pub/pmsi

[50] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore smp systems," in *IEEE/ACM International Symposium on microarchitecture (MICRO)*, 2009, pp. 413–422.

[51] M. E. Thomadakis, "The architecture of the nehalem processor and nehalem-EP SMP platforms," *Resource*, vol. 3, p. 2, 2011.

[52] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architure News*, vol. 39, no. 2, pp. 1–7, 2011.

[53] M. M. Martin, M. D. Hill, and D. A. Wood, "Token coherence: Decoupling performance and correctness," in *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, 2003, pp. 182–193.

[54] M. Lis, K. S. Shim, M. H. Cho, and S. Devadas, "Memory coherence in the age of multicores," in *IEEE 29th International Conference on Computer Design (ICCD)*, 2011, pp. 1–8.

[55] N. Agarwal, L.-S. Peh, and N. K. Jha, "In-network snoop ordering (inso): Snoopy coherence on unordered interconnects," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2009, pp. 67–78.

[56] M. Galles and E. Williams, "Performance optimizations, implementation, and verification of the sgi challenge multiprocessor," in *IEEE Hawaii International Conference on System Sciences*, 1994, pp. 134–143.

[57] V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *ACM Design Automation Conference (DAC)*, 2008, pp. 300–303.

[58] M. Schoeberl, W. Puffitsch, and B. Huber, "Towards time-predictable data caches for chip-multiprocessors," in *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 2009, pp. 180–191.

[59] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ ISSS)*, 2011, pp. 99–108.

[60] Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst case analysis of DRAM latency in multi-requestor systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2013, pp. 372–383.

[61] D. B. Kirk, "Smart (strategic memory allocation for real-time) cache design," in *IEEE Real Time Systems Symposium (RTSS)*, 1989, pp. 229–237.

[62] B. Lesage, I. Puaut, and A. Seznec, "Preti: Partitioned REal-TIme shared cache for mixed-criticality real-time systems," in *ACM Real-Time and Network Systems (RTNS)*, 2012, pp. 171–180.

[63] J. Bin, S. Girbal, D. G. Pérez, A. Grasset, and A. Merigot, "Studying co-running avionic real-time applications on multi-core COTS architectures," *Embedded real time software and systems (ERTS)*, 2014.

[64] A. Cortex, "Cortex-A9 MPCore," *Technical Reference Manual*, 2009.

[65] N. Kurd, J. Douglas, P. Mosalikanti, and R. Kumar, "Next generation intel® micro-architecture (nehalem) clocking architecture," in *IEEE Journal of Solid-State Circuits*, vol. 44, no. 4, 2009, pp. 1121–1129.

[66] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *IEEE International Symposium on Computer Architecture (ISCA)*, 1995, pp. 24–36.

[67] M. Schoeberl, "Time-predictable computer architecture," *EURASIP Journal on Embedded Systems*, vol. 2009, no. 1, p. 758480, 2009.

[68] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves for multimedia operating systems," DTIC Document, Tech. Rep., 1993.

[69] L. Abeni and G. Buttazzo, "Resource reservation in dynamic real-time systems," *Real-Time Systems*, vol. 27, no. 2, pp. 123–167, 2004.

[70] G. Buttazzo, E. Bini, and Y. Wu, "Partitioning real-time applications over multicore reservations," *IEEE Transactions on Industrial Informatics (IES)*, vol. 7, no. 2, pp. 302–315, 2011.

[71] Y.-S. Chen, H. C. Liao, and T.-H. Tsai, "Online real-time task scheduling in heterogeneous multicore system-on-a-chip," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 24, no. 1, pp. 118–130, 2013.

[72] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A predictable SDRAM memory controller," in *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007, pp. 251–256.

[73] B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration," in *IEEE Conference on Design, Automation and Test in Europe (DATE)*, 2011, pp. 1–6.

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

[75] S. Goossens, B. Akesson, and K. Goossens, "Conservative open-page policy for mixed time-criticality memory controllers," in *IEEE Conference on Design, Automation and Test in Europe (DATE)*, 2013, pp. 525–530.

[76] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and F. J. Cazorla, "A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study," in *IEEE Real-Time Systems Symposium (RTSS)*, 2014, pp. 207–217.

[77] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh, "A predictable and command-level priority-based dram controller for mixed-criticality systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015, pp. 317–326.

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

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

[80] S. Goossens, T. Kouters, B. Akesson, and K. Goossens, "Memory-map selection for firm real-time sdram controllers," in *IEEE Conference on Design, Automation and Test in Europe (DATE)*, 2012, pp. 828–831.

[81] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *IEEE International Symposium on Computer Architecture (ISCA)*, 2008, pp. 39–50.

[82] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens, "A reconfigurable real-time sdram controller for mixed time-criticality systems," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*. IEEE, 2013, pp. 1–10.

[83] M. D. Gomony, B. Akesson, and K. Goossens, "A real-time multichannel memory controller and optimal mapping of memory clients to memory channels," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, pp. 25:1–25:27, 2015.

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

[85] L. Ecco, A. Kostrzewa, and R. Ernst, "Minimizing dram rank switching overhead for improved timing bounds and performance," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2016, pp. 3–13.

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

[87] R. Kalla, B. Sinharoy, and J. M. Tendler, "Ibm power5 chip: A dual-core multithreaded processor," *Micro, IEEE*, vol. 24, no. 2, pp. 40–47, 2004.

[88] A. Radulescu, J. Dielissen, S. G. Pestana, O. P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens, "An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, pp. 4–17, 2005.

[89] B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems*. Springer, 2011.

[90] C. R. Houck, J. A. Joines, and M. G. Kay, "A genetic algorithm for function optimization : a matlab implementation," *NCSU-IE Technical Report*, 1995.

[91] J. Poovey, "Characterization of the eembc benchmark suite," *North Carolina State University, Technical Report*, 2007.

[92] "JEDEC DDR3 SDRAM specifications JESD79-3D." [Online]. Available: http://www.jedec.org/standards-documents/docs/jesd-79-3d

[93] P. Panda, N. Dutt, and A. Nicolau, "Exploiting off-chip memory access modes in high-level synthesis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1997, pp. 333–340.

[94] O. Mutlu and L. Subramanian, "Research problems and opportunities in memory systems," *Supercomputing frontiers and innovations*, vol. 1, no. 3, pp. 19–55, 2014.

[95] A. Abel and J. Reineke, "Measurement-based modeling of the cache replacement policy," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 65–74.

[96] W. Ding, J. Liu, M. Kandemir, and M. J. Irwin, "Reshaping cache misses to improve row-buffer locality in multicore systems," in *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 235–244.

[97] Y. Zhang, W. Ding, J. Liu, and M. Kandemir, "Optimizing data layouts for parallel computation on multicores," in *IEEE conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011, pp. 143–154.

[98] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 155–166.

[99] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 225–236.

[100] A. Shafiee, A. Gundu, M. Shevgoor, R. Balasubramonian, and M. Tiwari, "Avoiding information leakage in the memory controller with fixed service policies," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 89–101.

[101] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, 2014, pp. 361–372.

[102] T. John and R. Baumgartl, "Exact cache characterization by experimental parameter extraction," in *International Conference on Real-Time and Network Systems (RTNS)*, 2007, pp. 65–74.

[103] J. Dongarra, S. Moore, P. Mucci, K. Seymour, and H. You, "Accurate cache and TLB characterization using hardware counters," in *Springer International Conference on Computational Science (ICCS)*, 2004, pp. 432–439.

[104] C. L. Coleman and J. W. Davidson, "Automatic memory hierarchy characterization," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2001, pp. 103–110.

[105] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010, pp. 235–246.

[106] K. Yotov, K. Pingali, and P. Stodghill, "Automatic measurement of memory hierarchy parameters," *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1, pp. 181–192, 2005.

[107] C. Thomborson and Y. Yu, "Measuring data cache and TLB parameters under Linux," in *IEEE International Sympoisum on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, 2000, pp. 383–390.

[108] H. Park, S. Baek, J. Choi, D. Lee, and S. H. Noh, "Regularities considered harmful: forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 181–192, 2013.

[109] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Reverse engineering intel dram addressing and exploitation," *arXiv preprint arXiv:1511.08756*, 2015.

[110] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *Computer Architecture Letters (CAL)*, vol. 10, no. 1, pp. 16–19, 2011.

[111] Intel, "Intel Xeon Processor X5650." [Online]. Available: http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI

[112] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[113] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *ACM/IEEE international symposium on Microarchitecture (MICRO)*, 2000, pp. 32–41.

[114] W.-F. Lin, S. K. Reinhardt, and D. Burger, "Reducing DRAM latencies with an integrated memory hierarchy design," in *International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2001, pp. 301–312.

[115] Intel, "Intel Xeon Processor E7 family Uncore Performance Monitoring programming guide," 2011. [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-e7-family-uncore-performance-programming-guide.html

[116] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters (CAL)*, vol. 15, no. 1, pp. 45–49, 2011.

[117] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "Usimm: the utah simulated memory module," *University of Utah, Tech. Rep*, 2012.

[118] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX Security Symposium*, 2007, pp. 1–18.

[119] J. Millen, "20 years of covert channel modeling and analysis," in *IEEE Symposium on Security and Privacy (SP)*, 1999, pp. 113–114.

[120] U. Xilinx, "Ml505/506/507 evaluation platform user guide," *Document Revision*, vol. 3, no. 2, 2011.

[121] Micron, "Micron DDR2 SDRAM."

[122] A. Cosoroaba, "Memory interfaces made easy with xilinx fpgas and the memory interface generator," *Xilinx Corporation, white paper*, vol. 260, 2007.

[123] I. Corporation, "Intel 64 and ia-32 architectures, software developers manual, instruction set reference, a-z," 2011.

[124] "Intel platform and component validation , a white paper," http://download.intel.com/design/chipsets/labtour/PVPT_WhitePaper.pdf, Intel, 2015-08-31.

[125] H.-M. Koo and P. Mishra, "Test generation using sat-based bounded model checking for validation of pipelined processors," in *ACM Great Lakes symposium on VLSI (GLSVLSI)*, 2006, pp. 362–365.

[126] M. Katelman, J. Meseguer, and S. Escobar, "Directed-logical testing for functional verification of microprocessors," in *ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2008, pp. 89–100.

[127] M. Ghasempour, J. Garside, A. Jaleel, and M. Luján, "Dream: Dynamic re-arrangement of address mapping to improve the performance of drams," in *International Symposium on Memory Systems (MEMSYS)*, 2016, pp. 362–373.

[128] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: a DRAM page-mode scheduling policy for the many-core era," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 24–35.

[129] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006, pp. 208–222.

[130] "Mcxplore." [Online]. Available: https://caesr.uwaterloo.ca/mcxplore/

[131] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *IEEE International Conference on Formal Engineering Methods*, 1998, pp. 46–54.

[132] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in computers*, vol. 58, pp. 117–148, 2003.

[133] M. Jeong, D. Yoon, and M. Erez, "Drsim: A platform for flexible DRAM system research," *Technical Report*, 2012.

[134] M. Jung, C. Weis, and N. Wehn, "DRAMSys: A flexible DRAM subsystem design space exploration framework," *IPSJ Transactions on System LSI Design Methodology (T-SLDM)*, vol. 8, pp. 63–74, 2015.

[135] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 128–138, 2000.

[136] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged memory scheduling: achieving high performance and scalability in heterogeneous systems," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 416–427, 2012.

[137] Micron, "DDR3 SDRAM Verilog model," 2012.

[138] D. Sahoo and M. Satpathy, "MSimDRAM: Formal model driven development of a dram simulator," in *IEEE International Conference on VLSI Design and International Conference on Embedded Systems (VLSID)*, 2016, pp. 597–598.

[139] K. Khalifa and K. Salah, "Implementation and verification of a generic universal memory controller based on uvm," in *IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2015, pp. 1–2.

[140] M. O. Kayed, M. Abdelsalam, and R. Guindi, "A novel approach for sva generation of ddr memory protocols based on tdml," in *IEEE International Microprocessor Test and Verification Workshop (MTV)*, 2014, pp. 61–66.

[141] Y. Li, B. Akesson, K. Lampka, and K. Goossens, "Modeling and verification of dynamic command scheduling for real-time memory controllers," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–12.

[142] T. Poddar, N. Ahuja, and N. Ali, "Smart way to memory controller verification: Synopsys Memory VIP," https://www.design-reuse.com/articles/38587/synopsys-memory-controller-verification.html, Synopsys.

[143] J. Callahan, F. Schneider, S. Easterbrook *et al.*, "Automated software testing using model-checking," in *SPIN workshop*, 1996.

[144] V. Okun, P. E. Black, and Y. Yesha, "Testing with model checker: Insuring fault visibility," *International conference on system science, applied mathematics & computer science, and power engineering systems (WSEAS)*, pp. 1351–1356, 2003.

[145] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2009.

[146] "VC Verification IP for DRAM Memory," https://www.synopsys.com/verification/verification-ip/dram-memory.html, Synopsys.

[147] "Ddr4 memory protocol analysis and compliance verification," http://literature.cdn.keysight.com/litweb/pdf/5991-1827EN.pdf?id=2295381, Keysight Technologies and FuturePlus Systems.

[148] A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and J. Schumann, "A unified methodology for pre-silicon verification and post-silicon validation," in *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2011, pp. 1–6.

[149] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcu, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *AI magazine*, vol. 28, no. 3, p. 13, 2007.

[150] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *Computer Aided Verification*. Springer, 2002, pp. 359–364.

[151] *Memory Controller Hub (MCH), A datasheet*, Intel, 2 2005.

[152] B. Jacob and D. Wang, "System and method for performing multi-rank command scheduling in ddr sdram memory systems," 2009, US Patent 7,543,102.

[153] M. Ghosh and H.-H. S. Lee, "Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007, pp. 134–145.

[154] I. X. Processor, "E5-2600 product family uncore performance monitoring guide," *Intel Corporation*, 2012.