

High Performance and Predictable Memory Request Handling for Safety-Critical Multicores

by

Zhuanhao Wu

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, 2025

© Zhuanhao Wu 2025

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: Francisco Javier Cazorla Almeida
Group Leader, Computer Architecture Operating System,
Barcelona Supercomputing Center (BSC)

Supervisor: Hiren Patel
Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo

Internal Members: Rodolfo Pellizzoni
Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo

Mahesh Tripunitara
Professor, Dept. of Electrical and Computer Engineering,
University of Waterloo

Internal-External Member: Trevor Brown
Associate Professor, Dept. of Computer Science,
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored; see the 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

Zhuanhao Wu was the sole author of Chapters 1, 2, and 7 which were written under the supervision of Professor Hiren Patel and were not written for publication.

Chapters 3, 4, 5, and 6 of this thesis are based on five published manuscripts in peer-reviewed conference proceedings and journal articles, which I co-authored and to which I made major contributions. The inclusion of the contents from these publications in this thesis has been approved by all co-authors. Each publication includes a statement of my contribution and indicates the chapter in which it is included.

1. (Chapter 3) **Zhuanhao Wu**, and Hiren Patel, “Predictable Sharing of Last-Level Cache Partitions for Multi-Core Safety-Critical Systems”, in Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC), 2022 [[155](#)]
 - Developed, designed and implemented the last-level cache sharing mechanism
 - Developed the timing analysis
 - Execution of the experiments
 - Wrote portions of the article
2. (Chapter 4) **Zhuanhao Wu**, Anirudh Kaushik, and Hiren Patel, “High Performance and Predictable Shared Last-level Cache for Safety-Critical Systems”, in ACM Transactions on Embedded Computing Systems (TECS), 2024 [[153](#)]
 - Developed, designed and implemented the last-level cache partitioning sharing mechanism
 - Developed the timing analysis
 - Execution of the experiments
 - Wrote portions of the article
3. (Chapter 4) **Zhuanhao Wu**, Anirudh Kaushik, and Hiren Patel, “ZeroCost-LLC: Shared LLCs at No Cost to WCL”, in IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS), 2023 [[152](#)]
 - Developed, designed and implemented the last-level cache partitioning sharing mechanism
 - Developed the timing analysis

- Execution of the experiments
 - Wrote portions of the article
4. (Chapter 5) **Zhuanhao Wu**, and Hiren Patel, “Consistency-aware and Predictable Memory Processing for Safety-critical Out-of-order Multicores”, accepted to IEEE 31st Real-Time and Embedded Technology and Applications Symposium (RTAS), 2025
- Developed, designed and implemented the mechanisms for supporting multiple outstanding memory requests, and for enforcing the memory consistency model
 - Developed the mechanized correctness proof of the proposed mechanism
 - Developed the timing analysis
 - Execution of the experiments
 - Wrote portions of the article
5. (Chapter 6) **Zhuanhao Wu**, Marat Bekmyrza, Nachiket Kapre, and Hiren Patel, “Ditty: Directory-based Cache Coherence for Multicore Safety-critical Systems”, in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2023 [\[151\]](#)
- Developed, designed and implemented the predictable directory-based cache coherence protocol
 - Developed the timing analysis
 - Execution of the experiments
 - Wrote portions of the article

Abstract

Multi-core platforms are becoming the de-facto compute platforms for safety-critical systems such as ones used in automotive and avionics systems.

Recent works improve on the timing predictability and performance of safety-critical systems by designing and implementing predictable cache coherence protocols for these multicores. Yet, their application is restricted because of a few restrictions in the system architecture. First, prior works assume a single-level cache hierarchy, where each core has a private cache and a main memory is shared by all cores. However, in practice, a multi-level cache hierarchy is used, where each core has a private cache and a shared last-level cache, enabling the performance improvement by reducing the average memory access latency. Second, prior works on predictable cache coherence protocols assume that each core can issue at most one outstanding memory request. With recent interests in adopting out-of-order cores in safety-critical systems, the restriction on the number of outstanding memory requests can limit the overall performance of the system.

This thesis aims to address these restrictions and improve the performance of safety-critical systems by designing and implementing predictable cache coherence protocols for multicore systems with a multi-level cache hierarchy and multiple outstanding memory requests. The thesis has three main contribution:

First, this thesis shows that the sharing of an inclusive last-level cache among in a multicore system can lead to unbounded worst-case latency for memory requests. We additionally show that when deployed with a 1S-TDM arbitration schedule, the worst-case latency of memory requests can be bounded; yet, this worst-case latency bound is still prohibitively large, including a multiplicative factor of the number of cache lines in the private caches. The large worst-case latency is result of the interaction between the back-invalidations and write-backs of private cache blocks when multiple cores accessing the same cache set in the last-level cache. To address this issue, we propose a novel technique that provides predictable performance for memory requests in a multicore system when they share a last-level cache partition. Our approach eliminate the multiplicative factor of the number of cache lines in the private caches in the worst-case latency bound.

Second, we observe that the large worst-case memory request latency under shared last-level cache questions the applicability of the inclusive last-level cache in safety-critical multicores. To this end, we propose a novel cache architecture called the ZeroCost LLC that provides predictable performance for memory requests in a multicore system when they share a last-level cache partition. The ZeroCost LLC completely eliminates the back-invalidation in the inclusive last-level cache by performing cache line relocation during

a conflict set access. The ZeroCost LLC enables the use of an inclusive last-level cache in safety-critical multicores where cores share data without incurring worst-case latency penalties.

Third, to further improve the performance, we aim at lifting the restrictions in prior works on coherent data sharing mechanism, where each core has at most one outstanding memory request. Cores that can issue multiple memory requests, such as out-of-order cores, can improve the performance of the system by allowing more memory requests to be issued in parallel. We observe that when multiple outstanding memory requests are enabled, the mechanism that enforces memory consistency model in the cores can lead to increased worst-case latency for memory requests. To address this issue, we propose a novel approach in the cache coherence protocol that provides predictable worst-case latency for memory requests in a multicore system when multiple outstanding memory requests are enabled.

Finally, we investigate scaling up the number of cores in a multicore system by incorporating a predictable directory-based cache coherence protocol, ditty. Ditty leverages the observation that the arbitration of coherence messages in the directory is the main source of unpredictability in the cache coherence protocol, and it deploys a predictable arbitration mechanism to provide predictable performance for memory requests in a multicore system.

Acknowledgements

I would not have completed my PhD without the support of the people around me. First and foremost, I would like to thank my supervisor, Professor Hiren Patel, for his invaluable guidance and support throughout my PhD journey. I greatly benefited from his supervision style, which focuses on training students to become independent researchers. He taught me how to think critically and stay organized, to pierce through the complexity of research problems and identify their core. I also learned from him how to communicate my research ideas and technical challenges clearly and effectively. I will always remember his advice that how we present should reflect how we think. I feel extremely fortunate to have been able to work on problems I am passionate about, and I am grateful for the freedom he gave me to explore my research interests. As I complete my dissertation, I feel confident in my research skills and excited to apply them in my future career.

I am grateful to my dissertation committee members, Dr. Francisco J. Cazorla, Prof. Trevor Brown, Prof. Mahesh Tripunitara, and Prof. Rodolfo Pellizzoni, for their time, valuable feedback, and constructive suggestions on my work.

I feel privileged to have had the opportunity to work with the amazing people in the CAESR lab. Special thanks to Anirudh Kaushik (now a professor!), with whom I collaborated on several projects, including one that became a chapter of this thesis; his guidance and support were invaluable to me throughout my PhD. I would also like to thank the members of the CAESR lab, past and present, for their support, friendship, and the many good times we shared discussing research and life: Paulos Tegegn, Artem Klashtorny, Xinzhe Wang, Marat Bekmyrza, Michael Gionet, Ryan Ellis, and Susmita Tadepalli.

Special thanks to Dino Shen Yang, who believed in me and supported me throughout my PhD journey and through the ebbs and flows of life.

Finally, I would like to thank my family for their unwavering support and the life lessons they have taught me. Special thanks to Xuemei Peng, Eddie Ng, and Andrew Lin for their lasting support and influence, without which I would not be who I am today. I also thank Min Peng and Yong Peng, who shaped my early years and cultivated my interest in science and technology. They taught me the importance of hard work, perseverance, and the value of education. Their dedication and sacrifices for my education have inspired me to pursue my dreams and strive for excellence.

Dedication

This thesis is dedicated to Guanghui Zhou and Shuhua Peng.

Table of Contents

Examining Committee	ii
Author's Declaration	iii
Statement of Contributions	iv
Abstract	vi
Acknowledgements	viii
Dedication	ix
List of Figures	xiv
List of Tables	xviii
List of Abbreviations	xix
1 Introduction	1
2 Background	8
2.1 Overview of multicore architecture	8
2.2 Hardware cache coherence	13
2.3 Memory consistency model	17

2.4	Related Works	20
2.4.1	Predictable hardware cache coherent data sharing	20
2.4.2	Memory consistency enforcement mechanism	22
2.4.3	Last-level cache management in safety-critical multicores	23
3	Predictable sharing of last-level cache partitions	25
3.1	Related works	26
3.2	System Model	27
3.3	WCL with shared partitions	29
3.3.1	An unbounded WCL scenario	29
3.3.2	One-slot TDM schedule	29
3.3.3	Key observations	30
3.3.4	WCL analysis for 1S-TDM schedule	33
3.3.5	Set sequencer: Lowering the WCL	37
3.4	Evaluation	38
3.4.1	Worst-case latency	38
3.4.2	Partition sharing and utilization	39
3.5	Summary	39
4	Eliminating back-invalidations for predicatble safety-critical multicores	40
4.1	System Model	42
4.1.1	Baseline Memory Hierarchy	42
4.1.2	Memory Hierarchy with LLC	43
4.2	Background	45
4.3	The Impact on WCL when Using a Shared LLC	47
4.3.1	Expository Setup	48
4.3.2	Observations and Insights	49
4.4	ZCLLC: Shared LLC at No Cost to WCL	52

4.4.1	ZCLLC: Micro-architectural Extensions	52
4.4.2	ZCLLC: Dynamic Memory Update Policy on Replacements	53
4.4.3	Implementing ZCLLC in the LLC Controller	54
4.4.4	Illustrative Example	57
4.4.5	Characterizing ZCLLC's Write Policy	59
4.4.6	Hardware and timing overheads of ZCLLC	59
4.5	ZCLLC's Correctness	60
4.5.1	Ensuring Vacancy Invariant in ZCLLC	60
4.6	ZCLLC-OPT: Optimize ZCLLC for performance and predictability	63
4.6.1	TDM arbitration in ZCLLC	64
4.6.2	Observations and solutions	64
4.6.3	Worst-case Latency Analysis for ZCLLC-OPT	67
4.7	Empirical Evaluation	69
4.7.1	Worst-case Latency	71
4.7.2	Performance	72
4.7.3	WriteBack-induced Main Memory Accesses	75
4.8	Related Works	77
4.9	Summary	78
5	Consistency-aware and Predictable Memory Processing for Safety-critical Out-of-order Multicores	80
5.1	Background	82
5.1.1	Predictable hardware cache coherence	82
5.1.2	Memory consistency model	83
5.1.3	Key takeaways	88
5.2	System Model	88
5.3	Key Intuition: Memory Consistency Model Violation and Correction with PPP	90

5.4	PPP: Hardware Approach to Correct Potential Memory Consistency Model Violations with DLR	95
5.4.1	Hardware implementation of DLR correction	96
5.4.2	Hardware implementation to delay violating stores	97
5.4.3	Concrete execution	99
5.5	Worst-case Latency Analysis	100
5.6	Empirical Evaluation	104
5.6.1	Verification of correctness	104
5.6.2	Worst-case latency	104
5.6.3	Performance	105
5.6.4	Discussion	106
5.7	Summary	107
6	Directory-based coherence protocol for predictable cache coherence	108
6.1	Background and related works	109
6.1.1	Hardware cache coherence mechanisms	109
6.1.2	Related works	111
6.2	System model	111
6.3	Directory-based cache coherence mechanism	114
6.3.1	L2 cache controller design	114
6.3.2	Coherence protocol: enabling predictable data sharing	115
6.4	WCL analysis for memory requests	116
6.5	Evaluation	119
6.5.1	Synthetic workloads	120
6.5.2	SPLASH-3 benchmarks	120
6.5.3	Resource Utilization	120
6.6	Summary	121
7	Conclusion and Future Works	123
	References	125

List of Figures

2.1	(a) An example multithreaded program running on a multicore system has two cores executing two threads from the program. (b) In the system architecture, two cores execute two threads from the program, where each core is equipped private caches; the private caches are connected to a last-level cache through an interconnect. (c) An example in-order pipeline has five stages. (d) An example out-of-order pipeline has an execution engine that is capable of dynamically scheduling instructions to the functional units. .	9
2.2	The internals of a 4-way set associative cache.	12
2.3	Illustration of MSI protocol	14
2.4	The initial cache state of a dual core system.	15
2.5	An execution of the example program with cache coherence.	16
2.6	The sb litmus test.	17
3.1	Overview of system model.	27
3.2	c_{ua} experiences an unbounded latency.	29
3.3	c_{ua} 's request to cache line X eventually completes.	30
3.4	Distance of core caching l_1 increases after s_{cua}^{t+1}	32
3.5	The observed WCL of SS , NSS and P	36
3.6	The execution time of synthetic workload with fixed total partition size. . .	36
3.7	An illustration of set sequencer.	37
4.1	System model of ZCLLC.	44

4.2	(a) ZIV operation in LLC to avoid back-invalidations. (b) ZIV operation in LLC on a hit to a relocated cache line.	46
4.3	Different approaches of incorporating ZIV in an LLC: (a) employing ZIV by itself results in unbounded WCL (b) ZIV with ROC (c) Lower WCL with memory update.	47
4.4	Illustrative example.	57
4.5	The examples show optimizations applied to ZCLLC, assuming a TDM schedule of $[c_1, c_2, c_3]$. (a) TDM arbitration used in ZCLLC introduces under utilization of the LLC. (b) Work-conserving arbitration removes slot level under-utilization. (c) Sub-slot optimization removes under-utilization occurred within a slot. (d) Main memory accesses are serialized. (e) Split-bus optimization removes under-utilization of the main memory. (f) Write buffer allows overlapping of UpdateMemory with other requests.	63
4.6	Write buffer implementation in ZCLLC.	67
4.7	The observed worst-case latencies are shown for the following configurations: GP, ROC, ZIV+ROC, and ZCLLC for 2, 4, and 8 cores when running SPLASH-3 benchmarks. Optimizations to these configurations are applied incrementally to all configurations in the order of NORMAL (no optimization), WC, SUBSLOT, and OPT. The observed worst-case latencies are in cycles.	71
4.8	The speedups are shown for the following configurations: GP, ROC, ZIV+ROC, and ZCLLC with 2, 4, and 8 cores when running SPLASH-3 benchmarks. Optimizations to these configurations are applied incrementally to all configurations in the order of NORMAL (no optimization), WC, SUBSLOT, and OPT. All speedups are against the GP-NORMAL configuration.	73
4.9	The average memory access latencies are shown for both GP and ZCLLC. Configurations equipped with all optimizations (GP-OPT and ZCLLC-OPT) consistently achieve lower memory access latencies compared to configurations without optimizations (GP-NORMAL and ZCLLC-NORMAL).	75
4.10	The fraction of WriteBack that causes UpdateMemory indicates how many replacements from L2 actually causes access to the main memory in ZCLLC-OPT; we set the LLC size to 256kB to ensure the UpdateMemory is stressed when processing WriteBack.	76

4.11	The number of main memory accesses are normalized to ZIV+ROC-OPT; we set LLC size to 256kB to ensure the UpdateMemory is stressed when processing WriteBack.	77
5.1	(a) The example shows results disallowed in SC and TSO. (b) Forwarding value from (i1) to (i2) causes execution result disallowed in SC.	83
5.2	System model for PPP assumes that N out-of-order cores and the LLC are interconnected with a split-bus.	88
5.3	(a) Illustration of DLR. (b) PPP postpones violating store. (c) final memory order is equivalent to sequential consistency (SC) and total store order (TSO). Memory operations are from Figure 5.1(a).	91
5.4	The working of PPP. (a) A concrete execution of the program in Figure 5.1(a). (b) miss status holding register (MSHR) is extended with the delay store table index field (DST ID). (c) The delay store table in the arbiter split-transaction bus arbiter. (d) The operation of PPP when an invalidating store broadcasts on REQ. (e) The operation of PPP when the invalidating store is no longer blocked.	95
5.5	PPP extends coherence messages to manage the delaying of violating store and to ensure correct value forwarding.	99
5.6	Timeline of a request.	101
5.7	Observed worst-case latencies of Serial, DUEPCO, GP, and PPP for 2, 4, and 8 cores are shown in (a), (b), and (c) respectively. Performance of 2, 4, and 8 cores configurations are shown in (d), (e), and (f) respectively. Baseline configuration is GP. The GMean indicates geometric mean.	103
5.8	(a) Observed WCL of 2 to 8 cores configurations with synthetic benchmarks. The dashed lines tag the WCL of PPP. (b) Speedups of 8-core PPP, FCFS-PPP, and DP-Retry normalized to GP. (c) Running SPLASH-3 benchmarks on 2, 4, and 8 cores configurations with single-threaded execution. Speedups are normalized to GP.	103
6.1	(a) MSI cache coherence protocol. (b) The detailed transitions between M and I	110
6.2	Ditty system model.	112

6.3	(a) Ditty L2 cache contains controller (Ctrl) and storage and has an interface with predictable arbitration. (b) Directory (Dir) interface eliminates PutAck . (c) PutAck introduces long latency. (d) Ditty protocol eliminates PutAck . . .	114
6.4	A memory request of c_{ua} undergoes the WCL.	117
6.5	Observed WCL in cycles.	120

List of Tables

3.1	Symbols.	28
4.1	Symbols and names used in proofs and analysis.	53
4.2	Description of routines used in ZCLLC operation.	55
4.3	Worst-case latency of various configurations. The worst-case latency accounts for the latency of both FFLM and WTLM.	69
4.4	Observed WCL (Obs.) and Analytical WCL (Analyt.) for different configurations (in cycles).	70
6.1	Coherence messages in L2 and LLC	113
6.2	Interconnect latencies (Intc. Lat.), observed worst-case latency (OWL), analytical worst-case latency (AWL), and resource utilization of Ditty and GPD _{Dir} . Resource utilization is reported in the number of flip-flops (FFs), lookup tables (LUTs), and block RAMs (BR).	122

List of Abbreviations

COTS commercially available off-the-shelf [2](#)

DST delay store table [97–100](#)

FCFS first-come-first-serve [106](#)

GPU graphic processing unit [2](#)

L1D\$ L1 data cache [27](#)

L1I\$ L1 instruction cache [27](#)

L2\$ L2 cache [27](#), [28](#)

LLC last-level cache [27](#), [28](#)

LLC last-level cache [89](#), [90](#), [101](#), [102](#)

LRU least-recently used [31](#)

MBTA measurement-based timing analysis [3](#)

MLP memory level parallelism [80](#)

MSHR miss status holding register [xvi](#), [89](#), [95](#), [97](#), [98](#), [100](#)

MSI Modified-Shared-Invalid [109](#), [124](#)

NoC network-on-chip [14](#), [108](#)

PHCC predictable hardware cache coherence [4](#)

QLT Queue Lookup Table [37](#)

SC sequential consistency [xvi](#), [83](#), [85–88](#), [91](#), [93](#), [106](#)

SQ Sequencer [37](#)

STA static timing analysis [3](#)

TSO total store order [xvi](#), [81](#), [83](#), [85–88](#), [91](#), [93](#), [94](#), [106](#)

WCET worst-case execution time [3](#)

WCL worst-case latency [80–82](#), [88](#), [107](#)

Chapter 1

Introduction

Safety-critical systems are systems whose failure can result in loss of life, significant property damage, or environmental harm. These systems are used in a wide range of applications, including automotive, aerospace, medical, and defense systems [97]. The development of safety-critical systems is challenging due to the need to ensure that the system operates correctly under all conditions. Constraints in different domains tend to differ for different domains of applications.

In domains such as avionics and automotive, computing systems must not only ensure correct results for *logical correctness*, but, the system must also complete these computations within strict timing constraints to ensure *temporal correctness*. Failure to complete the computations within the timing constraints may result in catastrophic consequences such as loss of lives. Compute systems that depend on both logical correctness and temporal correctness are called *safety-critical real-time systems*, and are the focus of this thesis.

Consider the motion planning application in autonomous driving vehicle [28, 31, 96] to realize semi or fully autonomous driving capabilities. One task of this application is to properly handle the car following situation, where the vehicle performing the task, named *ego vehicle*, must follow the vehicle in front of it while maintaining a safe distance. Note that the first vehicle in front of the ego vehicle drives in the same lane (*ego lane*) as the ego vehicle, and we refer to this vehicle as the *front ego lane vehicle*. The task receives information on the surrounding vehicles and controls the vehicle to follow the front ego lane vehicle, while maintaining its longitudinal safety distances on a high way. For the safety of the human driver, the task must not only decide the control actions to perform when the front vehicle changes its speed (functional correctness), but it must also perform these computations within a strict timing constraint to ensure that the vehicle does not

collide with the front vehicles (temporal correctness).

Safety-critical multicore systems. Multi-core computing platforms are increasingly popular in safety-critical systems to satisfy the demand for high computation capabilities imposed by the increasing complexity of applications [9, 23, 106, 161]. Multi-core computing platforms consolidate multiple central processing units (CPUs or cores) onto a single component, resulting in enhanced performance, and reduced power consumption. Compared to the federated architecture approach where major functionality is deployed on dedicated computing nodes, multi-core platforms offer an advantage on the size, weight, and power (SWaP) of the system; in the meantime, it provides higher performance and reliability compared to single core systems, which must rely on higher frequency to achieve the same performance.

According to a recent survey [5], more than 80% of surveyed industrial safety-critical systems include multi-core components, where commercially available off-the-shelf (COTS) are typically deployed. The use of COTS is driven by (1) the availability of such platforms, (2) the vendors' efforts for verifying their implementation and certifying them for safety-critical standards, such as ISO 26262, and (3) availability of robust software ecosystems that facilitate the deployment of the software applications. Embedded multi-processor system on chip (MPSoC) platforms, such as AMD Versal Adaptive SoCs [9], NVIDIA AGX Orin [108], and Renesas R-Car H3 [116] include multi-core computing units with other compute accelerators such as programmable logic, AI engines, and graphic processing unit (GPU). We refer the readers to [23] for a comprehensive surveys of COTS multi-core computing units in safety-critical systems.

COTS multi-core platforms typically deploy a variety of hardware mechanisms to boost the performance or programmability of the system in its memory hierarchy. Examples of such mechanisms include (1) *out-of-order* execution pipelines, which allow the processor to execute instructions, including memory instructions, out of their program order to improve performance, (2) *hardware cache coherence* mechanisms, which ensure that the data shared between the cores are consistent, and (3) *shared last-level caches*, which store frequently accessed data and instructions to reduce the latency of memory accesses. This thesis centers around the design of *last-level caches*, and *hardware cache coherence* for safety-critical systems; these components serve as the backbone of the memory hierarchy in multi-core systems, and are crucial for the performance and predictability of the system. Caches are small on-chip memory units that store frequently accessed data and instructions to reduce the latency of memory accesses. Industrial safety-critical multi-core systems also take advantage of the caches to improve the performance of the system.

Timing predictability. Timing analyses are important for safety-critical systems to

ensure that the system operates within the timing constraints by deriving or estimating the worst-case execution time (WCET). Static timing analysis (STA) is one of the most common methods to derive the WCET of a task in a safety-critical system. In STA, a task in an application is first analyzed with control-flow analysis to identify properties in the task, such as loop iteration and possible execution paths. The analysis then combines the execution path information with a processor behavior model to estimate the worst-case execution time [130]. Alternative to STA is the measurement-based approaches such as measurement-based timing analysis (MBTA), where the measured execution time replaces the processor-behavior analysis in static methods. The schedulability analysis takes the WCET estimation from STA or MBTA as an input and generates a schedule for the tasks of an application, or deems that some or all tasks are not schedulable [36]. One important aspect of the processor behavior model is its memory system behavior, where the worst-case latency of a memory request is used in the WCET estimation of a task. Our theme in the thesis involves bounding the worst-case latency of a memory request, which can be used in WCET estimation as characterized in [130].

Thesis focus and motivation. Prior approaches in designing high performance memory systems for safety-critical multi-core systems typically make conservative assumptions, or constraints the use of the hardware resources to achieve timing predictability, limiting the performance of the system. This thesis aims at removing some of these limitations on performance of existing approaches to handle memory accesses in safety-critical multi-core systems, while preserving strict timing predictability. The improved performance benefits the safety-critical multicores by enabling the use of more complex applications, or by improving the quality of service of existing applications. For instance, the higher performance results in lower average task execution times, thereby increasing schedulable slack [137, 139]. This slack can be reallocated to (1) execute non-critical tasks, (2) schedule background maintenance tasks, or (3) enhance the system’s quality-of-service for soft real-time workloads. Also, for systems such as mixed criticality, the slack provided by the performance benefits can be used for other noncritical tasks [137, 139]. We next examine two aspects of the memory system that are crucial for the performance of safety-critical multi-core systems: (1) the shared last-level cache, and (2) the predictable hardware cache coherence, and discuss the limitations of existing approaches.

1. *Last-level cache management in safety-critical multicores.* The shared last-level cache is a typical resource that introduces interference causing unpredictable memory access latency. For example, the sharing of the last-level cache is limited through hardware and software approaches, where portions of last-level cache is partitioned and locked down for a task [46, 47, 102]. However, these approaches require profiling of the application to determine the proper cache partitioning, and may not be suitable for applications with

dynamic memory access patterns [7]. This could end up with smaller effective cache sizes, and may not be able to fully utilize the cache capacity. In addition, such approaches rule out coherent data sharing with the last-level cache across cores. As the number of cores are increasing, the absence of data sharing limits the application that could benefit from data sharing across cores.

2. *Predictable hardware cache coherence.* Another aspect of the performance is to investigate performance enhancement using predictable hardware cache coherence (PHCC). PHCC allows cached accesses to the shared data, while maintaining the timing predictability of the memory accesses. PHCC typically involves two main components to achieve timing predictability. First, hardware-based access controls, such as predictable arbitration schemes, are used throughout the memory hierarchy to arbitrate the resources. For example, a time-division multiplexing (TDM) arbitration scheme is used to allocate slots for each core to access the main memory controller [63, 76]. Second, the cache coherence protocol is designed to ensure that the memory accesses are consistent with the hardware modifications. While this line of research has shown promising results, existing works typically adopt conservative assumptions, leaving performance optimizations on the table. For instance, most state-of-the-art works assume one outstanding requests with no last-level cache [57, 63, 76, 78].

To address the limitations on performance of existing approaches to handle memory accesses in safety-critical multi-core systems, while preserving strict timing predictability, this thesis raises the following question:

Thesis question: How can the cache hierarchy for safety-critical multicore systems be designed to optimize performance and scalability, particularly in hardware cache coherence-based predictable shared data communication, with timing predictability?

A holistic approach to memory system optimization.

In this thesis, we take the first steps to enable performance optimizations in the memory system of safety-critical multi-core systems, while preserving strict timing predictability. Our approach starts by examining the integration of shared inclusive last-level caches in bus-based multi-core systems (Chapter 3). We provide a worst-case latency bound for shared inclusive last-level caches, and identify the key contributors to the worst-case latency is the back-invalidations. Next, we propose a novel last-level cache design, ZeroCost-LLC, that eliminates the back-invalidation overhead in shared inclusive last-level caches (Chapter 4). We then investigate the use of multiple outstanding memory requests in predictable cache coherence mechanisms (Chapter 5). Here, we identify the tension between enforcing memory consistency model and providing worst-case latency guarantees for memory

requests. Finally, we present a scalable predictable cache coherence data sharing mechanism, Ditty, that ensures predictable arbitration between the forward network and private cache ports (Chapter 6). This thesis provides evidence for the following thesis statement:

***Thesis statement:** The performance and scalability of cache hierarchy of a hardware cache coherence-based predictable shared data communication can be advanced through innovative mechanisms that target interference mitigation, enable dynamic consistency, and adapt to future architectural challenges, while preserving strict timing guarantees.*

Thesis contributions. The thesis makes the following contributions:

1. **Bounding The Worst-case Latency for Shared Inclusive Last-level Cache Partitions (Chapter 3).** This thesis begins by investigating the integration of shared last-level caches (LLCs) in bus-based multicore systems. LLC accesses are subject to interference from other cores, leading to significant worst-case latency compared to bypassing the cache. Interference arises when multiple cores compete for the same cache set, triggering evictions and back-invalidations to maintain inclusivity. We identify four key contributors to worst-case latency: (1) arbitration schemes, (2) back-invalidation overhead, (3) write-backs of dirty cache lines, and (4) mismatch between broadcast memory requests and LLC processing order. Unconstrained arbitration schemes can cause unbounded latencies. We demonstrate that employing 1S-TDM (one-slot time-division multiplexing) bounds latencies, but introduces a dependency on the private cache size. Our first contribution addresses this with a hardware extension named the set-sequencer; set sequencer enforces that the request ordering of the same cache set complies with the broadcast order, eliminating the multiplicative factor and improving bounds. Despite being bounded, this worst-case latency is cubic with respect to the number of cores.
2. **Eliminating the Back-invalidation Overhead in Shared Inclusive Last-level Cache (Chapter 4).** Back-invalidation overhead remains a challenge for shared inclusive LLCs in bus-based systems. Even with work-conserving TDM arbitration, latency remains quadratic with core count. Compared to the worst-case latency guaranteed by state-of-the-art predictable cache coherence mechanisms, this is a significant gap. In our second contribution, we close this gap by introducing ZeroCost-LLC, a novel LLC design that eliminates back-invalidation. Instead of evicting conflicting cache lines, ZeroCost-LLC relocates them to alternative sets. Additionally, it dynamically schedules dirty cache line write-backs to ensure relocations are always feasible. Deployed with linear predictable cache coherence, ZeroCost-LLC guarantees a linear worst-case latency, significantly improving scalability.

3. **Enabling Multiple Outstanding Memory Requests for Predictable Cache Coherence Mechanisms (Chapter 5).** Existing predictable cache coherence mechanisms assume only *one* outstanding memory request per core, limiting performance and hindering compatibility with speculative out-of-order pipelines. Relaxing this constraint is one step towards higher performance, but this risks unbounded latency, or even worse, memory consistency model violations. Our third contribution addresses this issue with hardware extensions and protocol modifications that allow multiple outstanding requests while preserving the sequential consistency and total store order memory consistency models. Consistency is corrected dynamically only when potential violations are detected, and the store requests that may violate the memory consistency model are delayed. Our solution ensures bounded worst-case latency and achieves significant performance gains over single-request designs.
4. **Scalable Predictable Cache Coherence Data Sharing Mechanism (Chapter 6).** Directory-based cache coherence mechanisms rely on messages traversing a network-on-chip, introducing new challenges for guaranteeing worst-case latencies. Enforcing that the network-on-chip is predictable is not sufficient to guarantee the worst-case latency of memory requests. In the fourth contribution, we present Ditty, a scalable data-sharing mechanism that ensures predictability by merging write-back messages with demand requests and employing predictable arbitration between the forward network and private cache ports. Ditty achieves scalability and timing guarantees while maintaining efficient data sharing.

Thesis Structure. Chapter 2 describes the background on predictable hardware cache coherence, memory consistency model, and a brief overview of related works. Next, Chapter 3 to Chapter 6 are the main research contribution of the thesis. Chapter 3 presents a scheme for predictably sharing cache partitions among multiple cores, where we bound the worst-case latency of memory requests with including a shared inclusive last-level cache. Our results indicate that the back-invalidation is the key culprit for the large worst-case latency of memory requests. Chapter 4 leverages this observation in Chapter 3 to propose an inclusive last-level cache called ZeroCost LLC (ZCLLC). ZCLLC eliminates the back-invalidation overhead by using a cache line relocation mechanism in the last-level cache, and provides a mechanism towards integrating shared inclusive last-level cache with predictable without increasing the asymptotic worst-case latency of memory requests. Chapter 5 investigates another mechanism for boosting performance - enabling the use of non-blocking private caches in safety-critical multi-core systems where the cache is capable of processing multiple outstanding memory requests from the core. Such a mechanism is crucial to leverage the memory-level parallelism exposed by out-of-order pipelines. We identify that the

main challenge lies in providing the worst-case latency guarantees for memory requests in the presence of non-blocking caches, while ensuring that the memory consistency model is preserved. Finally, Chapter 6 presents a directory-based predictable cache-coherence data sharing mechanism named ditty. Ditty is a first step towards scalable predictable cache coherence for safety-critical multi-core systems. We conclude the thesis with Chapter 7 and discuss possible future works based on the research contribution.

Chapter 2

Background

This chapter provides the necessary background on multicore architecture, predictable hardware cache coherence, memory consistency model, and an overview of related works. We discuss the related works in detail based on the research contribution in Chapter 3 to 6.

2.1 Overview of multicore architecture

The thesis focuses on multicore platforms for safety-critical embedded systems, where multiple cores execute instructions in parallel from different threads to improve performance. We will focus on memory instructions and their interactions in the multicore system. Figure 2.1(a) shows an example C++ program with two threads running on a multicore system. Thread 1 (`thread1` function) first stores the value 1 to memory address pointed to by the variable `pb` and then reads the value from the memory address pointed to by the variable `pa`. The two memory accesses will be translated to memory instructions by the compiler as a store instruction followed by a load instruction, referred to as (i1) and (i2), respectively. The concrete memory instructions are dependent on the instruction set architecture and the compiler, and our discussion is agnostic to a specific instruction set architecture. Similarly, thread 2 (`thread2` function) first stores the value 1 to memory address pointed to by the variable `pa` and then reads the value from the memory address pointed to by the variable `pb`, which are translated to memory instructions (i3) and (i4), respectively. In this example, the two threads share two memory addresses A and B.

These threads are executed on cores in a multicore system, as shown in Figure 2.1(b).

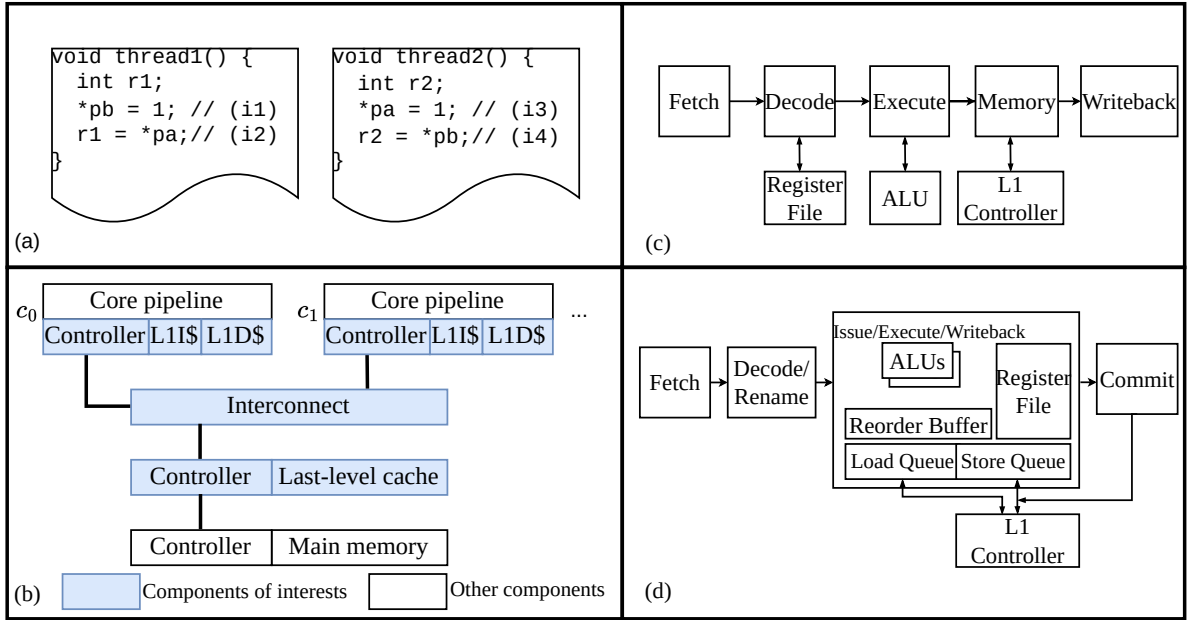


Figure 2.1: (a) An example multithreaded program running on a multicore system has two cores executing two threads from the program. (b) In the system architecture, two cores execute two threads from the program, where each core is equipped private caches; the private caches are connected to a last-level cache through an interconnect. (c) An example in-order pipeline has five stages. (d) An example out-of-order pipeline has an execution engine that is capable of dynamically scheduling instructions to the functional units.

Each core consists of a hardware pipeline that fetches instructions from the memory hierarchy, decodes them, and executes them. These pipelines are timing predictable, meaning that the execution time of an instruction is known a priori. To achieve better performance, the memory hierarchy consists of multiple levels of caches to exploit the temporal and spatial locality of memory accesses. This thesis focuses on the architecture where each core is equipped with a level of private instruction and data caches, and the private caches are connected to a shared last-level cache through an interconnect. The cache controllers at the private caches and the last-level cache are responsible for managing the accesses to the cache and ensuring that the data shared across the cores is coherent. The shared last-level cache is then connected to the memory hierarchy. The components of interests in this thesis are highlighted in blue.

Although this thesis focuses on the cache coherence mechanism, it is essential to understand the core pipeline design to understand the design principles of the cache coherence mechanism.

In-order pipelines. In-order pipelines enforce strict sequential processing of instructions, trading potential performance gains for simplicity and deterministic behavior. The in-order pipeline design is one of the popular designs in safety-critical and real-time systems, where timing predictable is important.

One canonical in-order pipeline follows a linear five-stage structure: instruction fetch, instruction decode, execute, memory access, and writeback, as shown in Figure 2.1(c). Instructions progress through these stages sequentially, with no reordering permitted. Dependencies between instructions are enforced through pipeline stalls, ensuring that each instruction completes its execution phase before subsequent dependent instructions proceed.

Memory operations in in-order pipelines adhere strictly to program order. Load (reads from memory) and store (writes to memory) instructions access the memory hierarchy sequentially. Store operations update the cache hierarchy immediately once sent to the cache hierarchy. In addition, at any given point in time, the pipeline has at most one outstanding memory operation. For its simplicity, it limits parallelism, as memory operations cannot overlap with each other, and subsequent instructions must wait for memory operations to complete before proceeding.

While the simplicity of in-order pipelines facilitates bounded worst-case execution times (WCETs), a critical property for real-time systems, this comes at the cost of underutilizing available hardware resources within the core as well as memory bandwidth [49, 53, 95].

Out-of-order pipelines. Modern multicores adopt out-of-order pipelines to enable high performance execution of instructions by dynamically reordering instructions while pre-

serving the logical correctness of programs. Figure 2.1(d) shows an example out-of-order pipeline with an execution engine that is capable of dynamically scheduling instructions to the functional units, using the famous Alpha 21264 processor as an example [79]. We use the Alpha 21264 processor as an example because it is the hardware that the widely adopted simulator gem5 [21] is based on. The gem5 simulator captures important micro-architecture features of modern processors such as multi-issue, register renaming and speculative execution, and it supports the performance evaluation of silicon-proven hardware designs [159]. The pipeline begins by fetching and decoding instructions sequentially in the instruction fetch and instruction decode stages. Decoded instructions then proceed to the register rename stage, where architectural registers are mapped to physical registers to resolve false data dependencies. If computational units (e.g., arithmetic logic units, floating-point units) or the load-store unit—a dedicated hardware component for managing memory operations—are unavailable, the pipeline stalls at this stage. The load-store unit plays a pivotal role in managing memory interactions. Load instructions and store instructions are buffered into separate load queues and store queues, respectively, in strict program order. These queues ensure that memory operations adhere to program semantics despite out-of-order execution. Loads and stores remain queued until their addresses are resolved and dependencies are satisfied. After renaming, instructions enter the issue/execute/writeback stage, where they execute out-of-order once their operands become available. Concurrently, all instructions are tracked in a reorder buffer, a first-in-first-out structure that guarantees in-order retirement by holding results until earlier instructions have completed. For memory operations, this involves additional constraints. For load instructions, they require resolved addresses and permission to access the cache hierarchy. And for store instructions, they calculate addresses and prepare data but defer cache updates until retirement to maintain consistency. Instructions await retirement in the commit stage, where they are finalized in program order. Store instructions are particularly sensitive to this process: A store is only transmitted to the cache subsystem after it is committed and all preceding stores in the store queue have retired. This ensures that memory updates become visible to other components strictly in program order, preserving the illusion of sequential execution while enabling parallelism.

Cache internals. Modern multicore processors employ multiple levels of caches to leverage the temporal and spatial locality of memory accesses. As illustrated in Figure 2.2, a cache typically consists of two primary components: the tag array and the data array. Both arrays are commonly implemented using static random-access memory (SRAM), which enables random access to memory locations via contiguous addresses. The data array stores the cached data, organized into fixed-size blocks known as cache lines. In modern processors, each cache line is typically 64 bytes. The tag array, on the other hand, stores

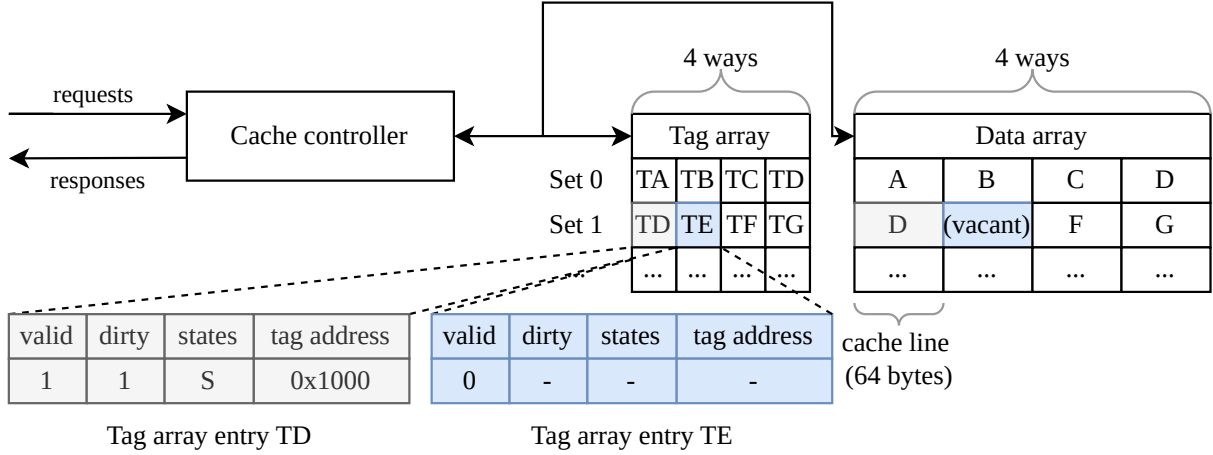


Figure 2.2: The internals of a 4-way set associative cache.

the address tags corresponding to the cached data, along with metadata such as valid and dirty bits. In multicore systems, the tag array also includes state bits to maintain cache coherence across cores, a topic that will be discussed in the next section.

The tag array enables the cache controller to determine whether a requested data block is present in the cache and to manage its state accordingly. Both the tag and data arrays are divided into sets, with each set containing multiple cache lines accessed in parallel. Sets are indexed using a portion of the memory address of the memory request, and the number of sets in the cache determines the associativity of the cache. A cache with an associativity of 1 is referred to as a direct-mapped cache, while a cache with associativity n (where $n > 1$) is termed an n -way set-associative cache. For example, Figure 2.2 depicts a 4-way set-associative cache. The four cache line entries in set 1 are not all filled. Tag entry TD, corresponding to the first data entry D, is valid and dirty, indicating that the data in D has been modified, and the address of the cached data is 0x1000. Tag entry TE corresponds to a vacant data entry, as indicated by the zero valid bit in the tag entry.

The cache controller orchestrates the operation of these components, managing memory requests from the core, coordinating with other caches, and interfacing with the broader memory hierarchy. On receiving a memory request, the cache controller first checks the tag array to determine if the requested data is present in the cache. If the requested data is found, the cache controller retrieves it from the data array and sends it back to the core for a load operation or updates the data array for a store operation. This process is known as a *cache hit*, otherwise, the data is not present in the cache, and this is a *cache miss*. On

a cache miss, the cache controller can fetch the requested data from the next level of the memory hierarchy, which may be another cache or main memory.

Caches can be generally classified based on their write policies and their allocation policies. The write policy determines how data is written to the cache and the backing storage (e.g., main memory) and is typically categorized into two types: *write-through* and *write-back*. In a write-through cache, the data is written to both the cache (if present) and the backing storage simultaneously. In a write-back cache, data is written to the cache first, and the backing storage is updated only when a dirty cache line is evicted due to set conflicts caused by a cache miss. The process of updating the backing storage is known as *write-back*. The write-back policy is typically used in modern processors due to its performance advantages, as it reduces the number of writes to the backing storage [72]. The allocation policy determines whether data is fetched into the cache when a cache miss occurs for a store. The two common allocation policies are *write-allocate* and *no-write-allocate*. In a write-allocate cache, the cache controller fetches the data from the backing storage into the cache before writing to it, while in a no-write-allocate cache, the data is written directly to the backing storage without fetching it into the cache. This thesis focuses on write-back caches with write-allocate policy, as they are more commonly used in modern processors. Depending on the cache level, the controller may handle requests from the core, private caches, or the last-level cache.

2.2 Hardware cache coherence

Hardware cache coherence is a data communication mechanism prevalent in multicore platforms that enables cores to simultaneously cache and access data in their private caches [105]. Cache coherence protocol is the key component in a hardware cache coherence mechanism and consists of rules to ensure the *single-writer-multiple-reader invariant* and the *data-value invariant* so that all cores observe the same value of a given datum at the same (logical) time. Recall the multi-threaded example in Figure 2.1(a) where two threads share two memory addresses A and B, hardware cache coherence ensures that the data written to B by (i1) of thread 1 are propagated so that thread 2's (i4) reads the updated value.

The key component in a hardware cache coherence mechanism is the cache coherence protocol, which defines a set of rules that maintain two variants. These rules are typically defined at the cache line granularity, typically 64 bytes in modern processors. A cache coherence protocol consists of two constructs: (1) **coherence states** that capture access permissions (read/write) of data, and (2) **state transitions** between coherence states that

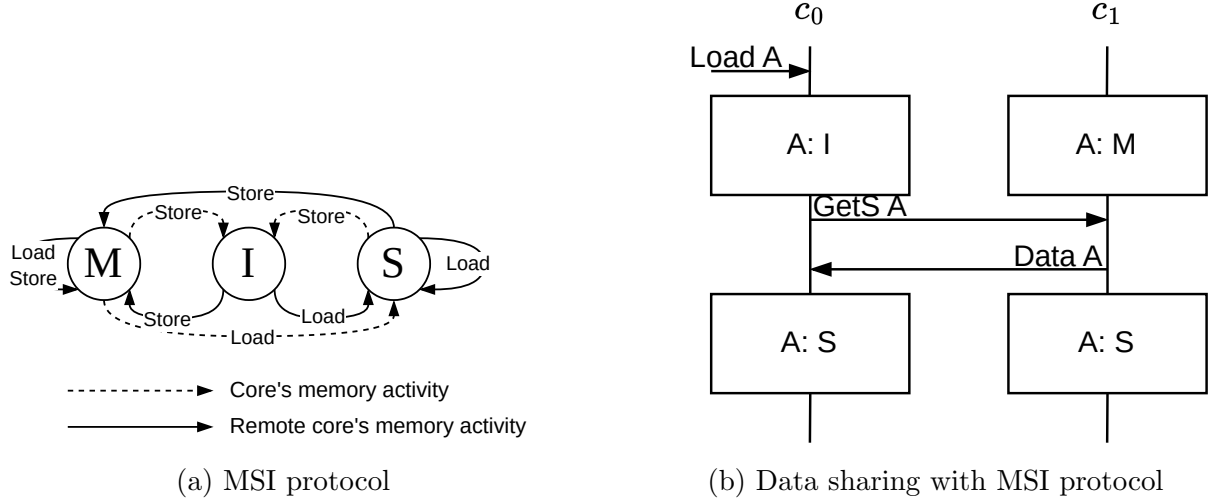


Figure 2.3: Illustration of MSI protocol

are triggered based on the memory activity of cores on the same data. Each core's cache controller implements the coherence protocol.

Figure 2.3 shows the Modified-Shared-Invalid (MSI) cache coherence protocol. Coherence protocols deployed in commercial multicore platforms build on top of the MSI cache coherence protocol [11, 33]. The MSI cache coherence protocol consists of three coherence states: **Modified (M)**, **Shared (S)**, and **Invalid (I)** [105]. A cache line in **M** state means that it has read and write access permissions, and its data contents have been modified. Only one core can have a cache line in **M** state at any time instance for data correctness [105]. A cache line in **S** state means that it has read access permissions, and its data contents have not been modified. Cores that have a cache line in **S** are referred to as *sharers*. A cache line in **I** state means that it is not available in the core's cache or its data contents are not up-to-date. State transitions between the coherence states are triggered on memory activity by cores on the same cache line. Based on the state transitions in MSI protocol (Figure 6.1a), c_1 changes its coherence state to **S** on observing c_0 's read request (Load), and c_0 moves from **I** to **S** state on receiving data for A; thereby allowing both cores to subsequently read A from their respective private caches.

Concrete implementations of cache coherence protocols typically involve additional states, named *transient states*, and transitions to optimize performance. Transient states enable the deployment of the protocol in a distributed manner on interconnect, such as a non-atomic snooping bus or a network-on-chip (NoC), where memory requests from mul-

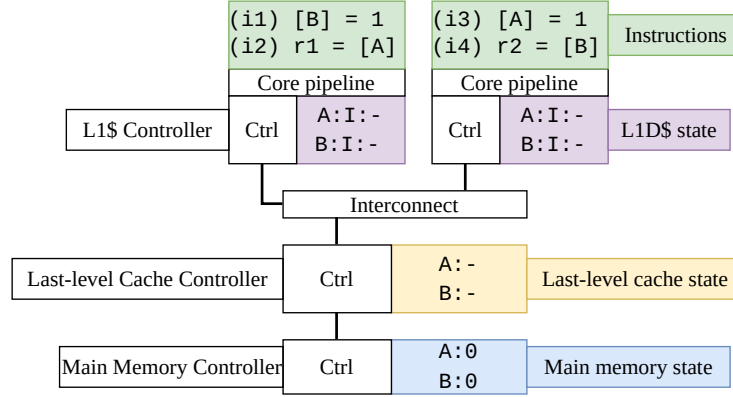


Figure 2.4: The initial cache state of a dual core system.

multiple cores interleave and the performance improves. For example, in MSI, when a reading core is waiting for a cache line to change from **I** to **S**, a non-atomic snooping bus allows for another core to request the same cache line in **M** before the cache line arrives the reading core. As a result, the final state of the reading core will be **I** instead of **S**. Adding transient states between **I** and **S** allows the reading core to account for such changes in the final coherence state [105].

We illustrate the working of the MSI coherence protocol with the cores executing the example multithreaded program in Figure 2.1(a). Figure 2.4 shows the initial state of the cores, their private caches, and the last-level cache. Initially, both c_0 and c_1 have the block A in **I** state, denoted as `A: I:-` in the L1 cache controller state. Similarly, B is in **I** state in both cores. The last-level cache is also empty. The main memory as the backing store of the cache hierarchy, stores the values of A and B, both being 0. We assume an in-order pipeline and the cores send their memory requests to the private cache in program order. Figure 2.5(a) illustrates the start of the execution. Core c_0 starts by executing (i1), which stores 1 to B. This causes c_0 to issue a store request `Store B` to the private cache. This request is a cache miss in the private cache; recall that a store request in the MSI protocol changes the state of the cache line to **M**. With a protocol with transient state, this transition cannot complete immediately, and the cache line is in a transient state, denoted as **IM^D**. This state indicates that the cache line is in the process of transitioning from **I** to **M**. Since the request misses in the private cache, the private cache controller must fetch the data from the memory hierarchy, which is the last-level cache, by broadcasting a `GetM` request to the last-level cache, which is also a cache miss. The last-level cache hence fetches the data from the main memory with a fetch request `Fetch B`. Figure 2.5(b) shows when the request for c_0 to B is inflight, c_1 executes (i3) concurrently, and stores 1

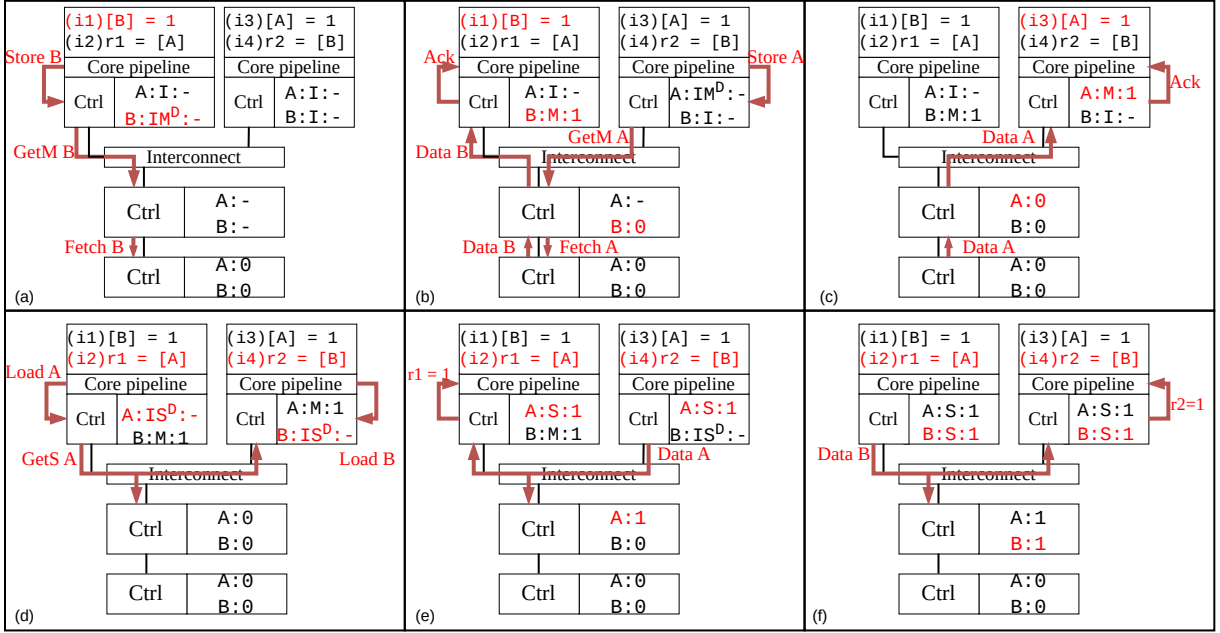


Figure 2.5: An execution of the example program with cache coherence.

to A . Similar to c_0 , c_1 's request for A is a cache miss in the private cache, and the cache line is in a transient state IM^D . And the private cache controller sends a **GetM** request to the last-level cache, which misses in the last-level cache and the last-level cache fetches the data from the main memory with a **Fetch** request. At the mean time, the data of B for c_0 's request is returned from the main memory to the last-level cache. The last-level cache controller fills its storage with the data for B and also sends the data to the private cache of c_0 . Upon receiving the data of B , the private cache controller updates the state of the cache line to **M** and sends the data to the core, concluding the memory request for (i1). The transient states that both c_0 and c_1 's private cache undergo enables the cache coherence protocol to handle the interleaving memory requests from multiple cores. c_1 's request to A completes similarly after the main memory returns the data of A to the last-level cache, and the last-level cache sends the data to the private cache of c_1 in Figure 2.5(c). One key role of cache coherence is to ensure that the data shared across the cores is consistent, and the MSI protocol ensures that the data is consistent by enforcing the single-writer-multiple-reader invariant and the data-value invariant. Figure 2.5(d)-(f) exemplify this process. In Figure 2.5(d), c_0 continues by executing (i2), which reads from A . This request is a cache miss in c_0 's private cache, and the private cache controller broadcasts a **GetS** request over the interconnect. Since c_1 caches A in **M** state, c_1 must

c_0	c_1
(i1) $[B] = 1$	(i3) $[A] = 1$
(i2) $r_1 = [A]$	(i4) $r_2 = [B]$
$r_1 = ?, r_2 = ?$	

Figure 2.6: The **sb** litmus test.

send the data to c_0 to ensure that c_0 observes the updated value of A. Meanwhile in c_0 , A's state transitions to **IS^D**, indicating that the cache line is transitioning from **I** to **S**. Similarly, c_1 executes (i4) concurrently, which reads from B, whose state transitions to **IS^D** in the private cache of c_1 . c_1 does not yet send a request to fetch the data yet due to internal delays. Next, in Figure 2.5(e), c_1 receives the **GetS** from c_0 and sends the data of A to c_0 over the interconnect. Note that c_1 also sends a copy of the data to the last-level cache to ensure that future requests from other cores observe the updated value of A. The state of c_1 's A transitions to **S** in the private cache of c_1 . Upon receiving the data of A, c_0 updates the data in the private cache and sends the data to the core. Hence, c_0 's (i2) reads the updated value of A into the register r_1 , which is 1. Finally, following the same process, c_0 sends the data of B to c_1 in Figure 2.5(f), and c_1 updates the data in the private cache and sends the data to the core.

2.3 Memory consistency model

While cache coherence determines how different cores observe changes within the same address, the memory consistency model (MC) defines how different cores observe changes across *different* addresses. Memory consistency model arises from the need to provide a consistent view of memory operations across different cores in a multicore system, when modern processors adopt aggressive optimizations, such as reordering memory instructions, that can produce unintuitive results. Specifically, the memory consistency model defines the *order* and *value* in which memory operations are observed by different cores in a multicore system.

A common way of specifying memory consistency model is through *litmus tests* [101]. Litmus tests are small programs that consist of a set of memory operations and synchronization primitives; execution results of litmus tests are used to reason about the memory consistency model. We use the **sb** litmus test in Figure 2.6 to illustrate the memory consistency model. The **sb** involves two cores c_0 and c_1 and two memory addresses A and B.

Initially, the content of A and B is 0. Core c_0 executes a store and a load instruction: (i1) writes 1 to B, and (i2) reads from A and set the value of register r_1 .; meanwhile in core c_1 : (i3) writes 1 to A, and (i4) reads from B and set the value in register r_2 . The *program order* is the order in which instructions are written in the program *per core*. Hence, (i1) is before (i2) in program order; (i3) is before (i4) in program order. After both cores complete their execution, there are multiple possible outcomes for the values of r_1 and r_2 depending on how the instructions are interleaved and ordered. For example, one possible outcome is that $r_1 = 0$ and $r_2 = 1$. Executing (i1) and (i2), then executing (i3) and (i4) entails this outcome. This order, in which all cores observe the memory operations, is called the *memory order*. For brevity, we denote this memory order as $\langle i1, i2, i3, i4 \rangle_m$. In **sb**, since (i3) and (i4) are the only instructions that modify A and B, there are only four possible outcomes for r_1 and r_2 , where r_1 and r_2 can be either 0 or 1. A memory consistency model specifies what are allowed outcomes, hence the memory order, for the litmus test and what are not. For brevity, we limit our discussion to the program order among Load and Store operations; hence there are four types of program order that need be specified by a memory consistency model: *load-to-load*, *load-to-store*, *store-to-load*, and *store-to-store*.

Sequential consistency. The most intuitive memory consistency model is the *Sequential Consistency (SC)* model [91]. The SC model enforces that the execution is as if the operations of all the processors were executed in some sequential order, and the operations of each individual core appear in this sequence in the order specified by its program [91]. This means that the memory order of the instructions must honor the program order of each core. For example, the aforementioned $r_1 = 0$ and $r_2 = 1$ outcome is allowed in the SC model, because in the memory order that produces this result, $\langle i1, i2, i3, i4 \rangle_m$, the program order of each core is respected. Similarly, the outcome $r_1 = 1$ and $r_2 = 0$ is allowed in the SC model, because in the memory order that produces this result, $\langle i3, i4, i1, i2 \rangle_m$, the program order of each core is respected. Note that multiple memory orders can produce the same outcome, and as long as there exists one memory order that respects the program order of each core, the outcome is allowed in the SC model. On the other hand, the outcome $r_1 = 0$ and $r_2 = 0$ is not allowed in the SC model because of the following reasoning. Since $r_1 = 0$, the load instruction (i2) must have read the value 0 from A, which is *before* (i3) writes to A. Hence (i2) must happen before (i3) in the memory order. Similarly, $r_2 = 0$ implies that the load instruction (i4) must have read the value 0 from B, which is the value *before* (i1) takes place. Hence (i4) must have happened before (i1) in the memory order. Additionally, since (i3) is before (i4) in c_1 's program order, (i3) must have happened before (i4) in the memory order. Therefore, combining these constraints, the final memory order is $\langle i2, i3, i4, i1 \rangle_m$. This memory order, however, violates the program order of c_0 , and hence the outcome is not allowed in the SC model.

Prior state-of-the-art predictable hardware cache coherence mechanisms for real-time systems, such as PMSI [58] and LPMSI [76] do not specify the memory consistency model they enforce, however, the fact that one core can have at most *one outstanding memory request* at a time implies that the memory consistency model is SC. Note that *a core having one outstanding memory request* means the private cache controller of the core can only process one memory request at a time, even if the core has multiple memory instructions in its pipeline.

Total store order. Another common memory consistency model is the *Total Store Order (TSO)* model [127]. TSO relaxes the store-to-load program order; this means that a load instruction can be executed before a store instruction that precedes it in the program order. TSO allows the outcome $r_1 = 0$ and $r_2 = 0$ in the **sb** litmus test. The memory order that produces this outcome is $\langle i3, i4, i1, i2 \rangle_m$. Since TSO does not enforce Store-to-Load ordering, the load instruction (i2) (rsp. (i4)) can be executed before the store instruction (i1) (rsp. (i3)) that precedes it in the program order. TSO allows a store buffer in the core that buffers store instructions and commits them to the memory in the order specified by the program order. Specifically, when a store is waiting in the store buffer, the core can execute subsequent load instructions, reducing stalls due to store-to-load ordering.

Tensions between ordering constraints, performance, and hardware design. Modern processors enforce memory consistency models stronger than what their underlying hardware can natively achieve, creating a delicate equilibrium between performance, hardware complexity, and software usability. For instance, while TSO architecturally only permits store buffers to hide write latency [127], the underlying hardware can reorder memory operations more aggressively to improve throughput. As part of the TSO requirement, cores must present the illusion of ordered load operations to software. To maintain this abstraction, hardware employs speculative execution: a stalled load (e.g., due to a cache miss or dependency) does not block younger loads, which may execute out-of-order speculatively. While this mitigates pipeline stalls, it risks violating the TSO if speculative state becomes visible globally, as in multicore systems where inter-core memory visibility is non-deterministic [45, 119]. These optimizations amplify challenges in scalable coherence protocols. For example, a speculative load in one core may read stale data from a cache line invalidated by another core’s write, violating TSO’s guarantee that loads observe the most recent store. To detect and resolve such violations, hardware mechanisms from both the core side and the cache side, such as invalidation-based tracking, in-order committing through reorder buffers, and pipeline squashing, must cooperate [45]. ARM architectures, with their weaker memory models [12], delegate more ordering enforcement to software via explicit barriers, reducing hardware overhead but complicating programming. Ultimately, the design of memory consistency models reflects a trilemma: strict models simplify cor-

rectness but limit parallelism; relaxed models improve performance but require intricate hardware/software co-design. As heterogeneous architectures (e.g., CPUs+GPUs, neuro-morphic chips) proliferate, balancing these trade-offs will demand novel paradigms, such as programmable consistency protocols or machine learning-guided speculation.

Incorporating high performance processors in safety-critical systems adds one more layer of complexity to the already complex tensions: on top of the hardware complexity, performance, and functional correctness requirements, safety-critical systems must also ensure that the system is predictable and that the system meets the safety requirements. We show in this thesis the possibility of reconciling these tensions by designing holistic solutions that address the performance, hardware complexity, and predictability requirements of safety-critical systems.

2.4 Related Works

2.4.1 Predictable hardware cache coherent data sharing

Predictable hardware cache coherence. Recent studies [58, 75] have made significant progress in developing predictable hardware cache coherence mechanisms for real-time systems. In [58], the authors propose a predictable hardware cache coherence mechanism designed for real-time systems, assuming a single-level cache hierarchy and a shared snooping bus. Their work establishes six critical invariants that guarantee bounded worst-case latencies for memory accesses in multicore systems. These invariants ensure that: (1) the shared snooping bus, which arbitrates access from private caches to main memory, employs a predictable arbitration scheme; (2) main memory processes memory requests strictly in the order of their arrival; (3) a core responds to requests from other cores in the same sequence as the requests are broadcast; and (4) a core’s write-back of a dirty cache line is predictably arbitrated alongside its memory requests.

To enforce these invariants, the authors introduce a set of micro-architectural modifications to the cache controller and propose a novel cache coherence protocol, termed PMSI (Predictable Modified-Shared-Invalid). This protocol builds upon the traditional MSI protocol but incorporates the aforementioned invariants to ensure predictability.

A subsequent study [75] extends the PMSI protocol by incorporating exclusive and owned states, which are commonly found in cache coherence protocols for general-purpose processors. These additions aim to improve performance without sacrificing predictability. The exclusive state allows a core to modify a clean cache line that is not shared with

other cores, eliminating unnecessary bus transactions. Similarly, the owned state enables multiple cores to share a dirty cache line without triggering write-backs to main memory, thereby reducing memory traffic.

However, PMSI and its extensions impose two key limitations: (1) each core is restricted to having only one outstanding memory request at a time, and (2) the worst-case latency of memory requests grows quadratically with the number of cores in the system. While these protocols demonstrate promising performance for real-time systems, their scalability and applicability to high-performance multicore systems—particularly those with larger core counts or greater memory-level parallelism—remain constrained by these drawbacks.

Tackling the quadratic worst-case latency with bus modifications. Later research works identify that the limitations of the quadratic worst-case latency growth in PMSI and its extensions are a result of a shared bus arbitrating two types of messages at the same time: the request messages from a core to the main memory controller, and the data response messages from the main memory controller to the core [63, 154]. In light of these observations, authors in [63] propose a split transaction bus architecture, named PISCOT, where a predictable TDM arbiter arbitrates the requests from the cores to the main memory controller, and a first-come-first-serve high performance arbiter that arbitrates the data responses from the main memory controller to the cores. PISCOT also involves a novel cache coherence protocol that ensures predictable worst-case latencies for memory accesses in multicore systems. On the other hand, the authors in [154] also propose a split transaction bus architecture that separates the concerns of request and response arbitration. To achieve predictability, the authors incorporate a dedicated data bus that allows parallel access between the main memory controller and the cores in addition to a shared request bus that adopts a predictable TDM arbitration scheme. The later work also validates the proposed architecture using a hardware prototype on an FPGA. In [76], authors propose a linear predictable cache coherence mechanism and assume the presence of cache-to-cache transfers. In addition, the authors propose a coherence protocol change where the owner of a cache line relinquishes the ownership eagerly on another core’s load request. These optimizations eliminate the queuing of write back latencies from the worst-case latency of memory requests in PMSI, and hence the worst-case latency is linear with respect to the number of cores in the system. All these works successfully address the quadratic worst-case latency growth in PMSI and its extensions by modifying the shared bus architecture, achieving a worst-case latency that is linear with respect to the number of cores in the system.

Multiple outstanding requests DUEPCO. A recent work, DUEPCO [103], relaxes the constraint on the number of outstanding memory requests per core, allowing multiple outstanding requests from a single core. This is achieved through a two-step arbitration

mechanism on the shared bus: the request bus employs a work-conserving round-robin arbitration, while the response bus utilizes a dynamic priority-based arbitration. The worst-case latency bound is quadratic in the number of cores, under the assumption that each core has at most one outstanding request for any given cache line. However, cores can still issue multiple outstanding requests to different cache lines.

Despite its performance advantages over other predictable cache coherence mechanisms, DUEPCO does not specify the memory consistency model it enforces. Consequently, naively applying DUEPCO to real-time systems can lead to two potential issues. First, if the core pipeline does not enforce any memory consistency model, the memory consistency model provided by the cache coherence mechanism may be insufficient to ensure program correctness. Second, if the core pipeline implements a memory consistency model, its interaction with the cache coherence mechanism may result in an enlarged worst-case latency bound, as guaranteed by DUEPCO. This could lead to an underestimation of the system’s worst-case execution time.

2.4.2 Memory consistency enforcement mechanism

Out-of-order pipelines for SC and TSO can reorder the execution of load instructions speculatively. This means that a load instruction that is later in the program order can be executed before a load instruction that is earlier in the program order. Recall that load instructions are ready to be sent to the cache in the memory hierarchy when the address calculation completes, the cache controller hence can receive the load instructions out-of-order. If store instructions from other cores that are executed between the load instructions, the store instructions can update the data that the load instructions are supposed to read, violating the load-to-load ordering memory consistency model. Hence, multicores supporting load reordering must deploy hardware mechanisms to enforce load-to-load ordering to ensure that the program’s correctness is not violated.

A prior work studies the enforcement of load-to-load ordering when load instructions are executed speculatively [45]. In this approach, when a load instruction is executed speculatively, and when a store instruction from another core updates the address that the load instruction is supposed to read, the load instruction is marked as to be retry. When this instruction is later committed, the load instruction is re-executed to ensure that the data read is correct. A similar approach is adopted in the Alpha 21264 processor [79], which is the hardware that the widely-adopted simulator gem5 is based on. Since the retry of load instructions is managed by the core pipeline, the cache controller is oblivious to the retry of load instructions; the decoupling of the core pipeline and the cache controller facilitates the implementation of the cache controller.

Another approach to enforce load-to-load ordering involves cooperation between the core pipeline and the cache controller [119]. This approach does not use the retry mechanism to enforce load-to-load ordering, when a speculatively executed load instruction sees a store instruction from another core to the requested address. Instead, the store instruction is delayed until the load instruction is committed. The cache controller hence must be capable of stalling the store instruction; this approach must also handle any potential deadlocks that may arise from the store instruction waiting for the load instruction to be committed, and any memory requests arriving when the store instruction is stalled.

Finally, in processors adopting relaxed memory orders, such as ARM [12] and RISC-V [146], the enforcement of load-to-load ordering is managed by the programmer. This is achieved through the use of explicit memory fences, which are instructions that enforce a specific ordering of memory operations. However, this approach places the burden of ensuring correct memory ordering on the programmer, which can lead to increased complexity and potential errors in software design. Additionally, some pipeline implementations such as the BOOM core [166] implement fences by *serializing* the memory operations, which can lead to performance degradation.

To the best of our knowledge, none of the prior works on predictable hardware cache coherence mechanisms for real-time systems discuss the load-to-load ordering enforcement.

2.4.3 Last-level cache management in safety-critical multicores

The shared last-level cache in multicore architectures is a well-documented source of inter-core interference, particularly in safety-critical real-time systems [46, 51, 102]. This interference arises when concurrent accesses from multiple cores contend for shared cache resources, introducing timing unpredictability that is unacceptable in domains requiring stringent worst-case execution time guarantees. Historically, modeling and statically analyzing this contention have posed significant challenges [140], primarily due to the dynamic and state-dependent interactions between cores during cache access. To address this, researchers have focused on isolation mechanisms that eliminate or tightly bound inter-core interference. Among these, cache partitioning—allocating fixed, non-overlapping portions of the LLC to individual cores or tasks—has emerged as a widely adopted solution. Cache partitioning can be implemented via software-based or hardware-assisted approaches, each with distinct trade-offs in flexibility, overhead, and compatibility.

The most prevalent software-driven technique for cache partitioning is page coloring, which exploits the virtual-to-physical address translation mechanism in modern operating systems. Page coloring leverages the property of physically indexed LLCs, where specific

bits of a physical address (the "color") determine the cache set index. By assigning virtual memory pages with distinct colors to different cores or tasks, the operating system ensures that their memory regions map to disjoint cache sets, thereby preventing overlap. This isolation enables hard real-time systems to achieve predictable cache behavior, which can then be integrated into schedulability analyses. Prior work has combined page coloring with (1) cache-aware scheduling algorithms that co-optimize task execution order and cache allocation [51], and (2) partition allocation algorithms that derive color assignments to minimize WCET while satisfying task constraints [102]. However, page coloring requires OS support and imposes constraints on virtual memory management, complicating its adoption in legacy or resource-constrained systems.

In contrast to software-based methods, hardware cache partitioning relies on dedicated circuitry to enforce spatial or temporal isolation. Modern architectures often provide partitioning features such as way-based division (e.g., Intel Cache Allocation Technology [67]) or address-range registers that restrict cacheable memory regions per core. These mechanisms enable finer-grained control over cache allocation, often with lower runtime overhead compared to software approaches. However, hardware partitioning still necessitates coordination with higher software layers: the operating system or application must configure partition boundaries, and real-time task models must account for partition sizes during WCET analysis. For instance, Suhendra et al. [140] propose a framework where task-specific cache quotas are communicated to hardware via OS-level APIs, while Arm's DynamIQ Shared Unit (DSU) [10] demonstrates how hardware-software co-design can enable dynamic partition reconfiguration. Despite these advancements, hardware partitioning solutions remain limited by their dependence on specialized hardware support, which may not be available in all platforms.

While cache partitioning effectively bounds inter-core interference, it introduces trade-offs in cache utilization efficiency. Partitioning can lead to underutilization of shared cache resources, as cores may not fully occupy their allocated cache space, particularly in asymmetric workloads or when tasks have varying cache requirements. This further complicates the deployment of cache partitioning as it requires careful strategies to balance isolation and resource utilization [84, 102], especially in safety-critical systems where efficient cache management is crucial for meeting stringent timing constraints.

Chapter 3

Predictable sharing of last-level cache partitions

Last-level cache optimizes the performance of a multicore system by caching and servicing data that are frequently accessed by the cores. It is a key component in delivering high performance in general-purpose processors. As mentioned in Section 2.4.3, the shared nature of the last-level cache complicates the worst-case timing analysis necessary for safety-critical systems due to the complex inter-core temporal interference [140].

LLC partitioning is a countermeasure to address these difficulties [43, 99, 122]. LLC partitioning allocates a part of the LLC to each core that it can use. This provides temporal isolation to tasks executing on a core from execution of other tasks on other cores. However, there are multiple downsides to LLC partitioning: (1) it can significantly affect average-case performance, (2) it can lead to underutilization of cache capacity, and (3) prevent coherent data sharing [75]. Downside (1) is a result of each core having a smaller part of the LLC. (2) happens when a core gets allocated a partition that it does not effectively use. For (3), conventional LLC partitioning disallows one core to access another core's partition; thus, coherent caching of shared data in the LLC is prohibited. With the continued increase in demand for functionalities, and their consolidation onto a multicore platform, we expect these downsides to overwhelm the benefits of LLC partitioning, necessitating the sharing of LLC partitions.

While prior works speculate that the worst-case latency of accessing a shared LLC partition is difficult to derive [46, 140], whether it is bounded or not remains unknown. In this chapter, we take a cautious step towards addressing these downsides and address the question: *what is the worst-case latency of memory accesses from cores that miss in their*

private caches, and access the shared partition?

We show that the caution in prior works is warranted; the worst-case latency of accessing a shared LLC partition can be unbounded: naively arbitrating cores' accesses to the shared LLC partitions results in a scenario where the WCL is unbounded. Nevertheless, we can correct this unbounded scenario by using a restricted version of time-division multiplexing (TDM) policy called 1S-TDM that results in a WCL bound. However, the resulting WCL bound is grossly pessimistic; it is proportional to the minimum of the cache capacity and LLC partition size of a given core and cube of the number of cores. By analyzing the critical instance, we intuit a technique to significantly lower the WCL. This technique yields a WCL that eliminates the dependency on the cache and partition sizes. For a 4-core setup with a 16-way LLC with 128 cache lines, our approach results in a WCL that is 2048 times lower. We implement this technique in a hardware structure called the set sequencer. We show that careful sharing of cache partitions not only allows for a low WCL, but possibly higher average-case performance. We envision the proposed work to complement existing efforts on LLC partitioning where certain tasks have their own partitions, but others share partitions; all of which depends on their performance and real-time requirements. Our main contributions in this chapter is as follows.

- We identify that naively using TDM to arbitrate accesses to the shared LLC partitions can result in an unbounded WCL. We establish that a 1S-TDM schedule prohibits this scenario.
- We develop a WCL analysis for a memory access to a shared LLC partition using the 1S-TDM arbitration policy.
- We propose a hardware extension called the set sequencer that lowers the WCL when sharing a LLC partition. We evaluate the set sequencer using simulation.

3.1 Related works

Cache partitioning [46] reserves a portion of the cache to a task or a core either via hardware or software techniques [156]. The key role of cache partitioning is to improve temporal isolation to simplify the WCL analysis. However, as the number of cores increase, allocating distinct partitions to each core or each task can affect average-case performance and cache under-utilization. Moreover, this prohibits deploying a large number of functionalities on the multicore as it may result in extremely small partitions to each functionality, which would adversely affect performance. The proposed work seeks a middle ground where

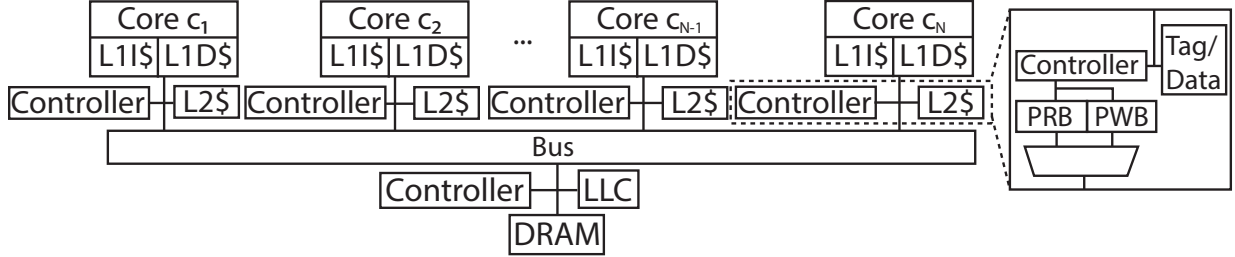


Figure 3.1: Overview of system model.

designers can judiciously share partitions with a subset of cores, and isolate others. Prior works identified that accurately capturing the contention of multiple cores sharing the LLC is difficult [140], and attempts exist for shared cache for dual-core processor relying on knowledge of the application [142]. Our work does not rely on application-specific knowledge and does not constraint the number of cores. Our work also applies when all cores share the whole LLC which serves as a single partition.

Authors in [70] noticed that interference between tasks exists when multiple tasks or cores share the LLC. They proposed a time analysable shared LLC where the inter-task interference is bounded probabilistically. Given their proposed technique, the exact worst-case interference in the LLC and the exact worst-case latency is not discernible. Compared to [70], our analysis provides an exact (non-probabilistic) bound of a memory access in the LLC, and does not rely on MBPTA. Our work assumes that a TDM bus arbitration has one slot per core in each period, which is common in controlling access to resources in safety-critical systems [52, 75]. We also identified the worst-case scenario for the LLC evictions, and that in the worst-case the latency can be unbounded if the arbitration has no constraint. Moreover, our analysis does not rely on certain type of address mapping or replacement policy.

3.2 System Model

System hierarchy. We assume a system with N cores and three levels of cache in the memory hierarchy (Figure 3.1). Each core has a private L1 instruction cache (L1I\$) and a private L1 data cache (L1D\$). The private L1 caches of a core are connected to a L2 cache (L2\$). A set-associative last-level cache (LLC) connects to all L2 caches, and interfaces with a DRAM directly. The accesses from L2 to LLC is controlled by a shared bus. The bus implements a time-division multiplexing (TDM) arbitration which allocates slots of

Table 3.1: Symbols.

Symbol	Explanation	Symbol	Explanation
c_i, c_{ua}	Core i , core under analysis	$d_{c_i}^{c_i}, d_{c_j}^{c_j}(x)$	Distance between cores
l_i, l	Cache lines in LLC set	SW	Slot width
N, n	# cores, # cores sharing a partition	$set_{LLC}(X)$	Cache set in the LLC that X maps to
$s_{c_{ua}}^t$	Start of the t -th slot of core c_{ua}	$m_{c_{ua}}, M$	Cache capacity of c_{ua} /LLC partition
w	# of ways in a partition	$\mathcal{P}$	Cache partition

width SW to cores. Table 3.1 shows the symbols used. SW is long enough for a data transfer between the L2\$ and the LLC. The L2 cache controller of each core only accesses the LLC in the core’s allocated slot, and the LLC only responds to the L2 within the core’s slot. Since the L2 cache is private to each core, a core accessing the bus implies that the core’s L2 cache controller access the bus. An LLC partition isolates a part of the LLC for a specific core. One task can be mapped to one core.

Similar to prior works [75], we assume each core has at most one outstanding memory request. Before a core’s request or write-back is placed on the bus, the request is buffered in a pending request buffer (PRB), and the write-backs are buffered in a pending write-back buffer (PWB). A predictable arbitration such as round-robin between PRB and PWB chooses from a request or a write-back to send on the bus at the beginning of the core’s slot.

Cache inclusion policy. We assume that the LLC is inclusive of L2. This is a common setup in existing platforms. Suppose that core c_i ’s request to the cache line at address X is a miss in all its private caches, and the LLC. Then, for the LLC to respond with the provided data for X , the LLC must ensure: (1) that there is a vacant entry in the set that cache line X maps to, (2) the cache line from a lower level memory in the memory hierarchy is fetched, and (3) the response to c_i is sent in c_i ’s slot. An important property of inclusive caches is that an eviction of a cache line in the lower-level cache requires eviction of cache lines for the same address in upper-level caches. For our setup, an eviction in the LLC forces evictions in both the L1 and L2 private caches that have the data.

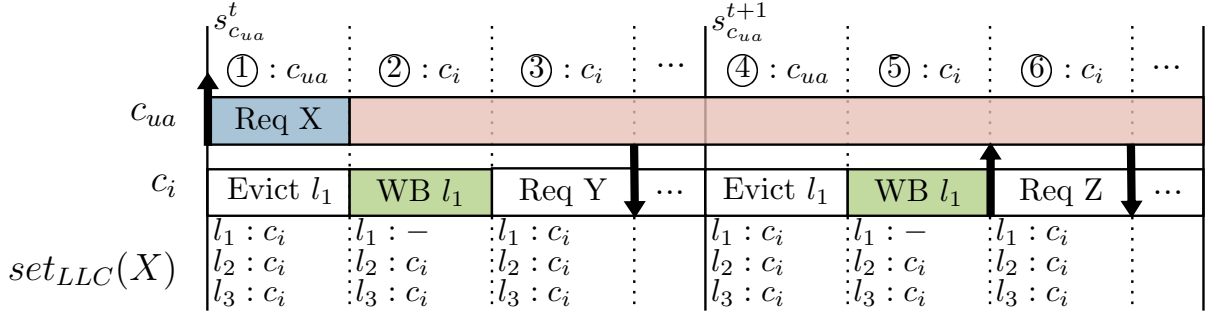


Figure 3.2: c_{ua} experiences an unbounded latency.

3.3 WCL with shared partitions

3.3.1 An unbounded WCL scenario

We show that an undesired consequence of the inclusive property, and multiple cores making accesses to the LLC is a situation where the WCL is unbounded. Using Figure 3.2, we illustrate this scenario. We assume a TDM arbitration policy with one slot for c_{ua} and two slots for c_i . We use $s_{c_i}^t$ to denote the starting time of the t -th slot of core c_i . Unambiguously, $s_{c_i}^t$ also refers to the t -th slot of c_i . At $s_{c_{ua}}^t$, in ①, c_{ua} 's request to a cache line X misses in the private caches and the LLC. Cache line X is mapped to a full cache set $set_{LLC}(X)$ in the LLC; thus, the LLC evicts a cache line l_1 in $set_{LLC}(X)$, which is also privately cached by c_i denoted by $l_1 : c_i$. Note that a full cache set means that there are no empty cache lines in the set. In ②, c_i writes back l_1 , and frees an entry in $set_{LLC}(X)$. Then, in ③, c_i requests a cache line mapped to $set_{LLC}(X)$, and gets the response within its slot because l_1 is available. In c_{ua} 's next slot $s_{c_{ua}}^{t+1}$ (④), $set_{LLC}(X)$ is full again preventing c_{ua} from completing its request. This behavior can potentially repeat indefinitely, which shows that the interference at the LLC from other cores can cause unbounded WCL for the core under analysis. Note that allowing the write-back within the same slot as the request does not mitigate the issue because another core can issue a request between the write-back and the response, occupying the freed entry.

3.3.2 One-slot TDM schedule

The scenario where c_{ua} experiences unbounded latency happens when a core other than c_{ua} is allowed to access the LLC multiple times before c_{ua} can access the bus again. An

effective solution to prevent another core from occupying a free entry in $set_{LLC}(X)$ is to constrain the TDM schedule to allocate only one slot per core in a period (Definition 1). Note that the crux of the issue is that an entry freed by c_i due to the eviction of cache line l_1 is occupied again by c_i again *before* c_{ua} can access the LLC. With 1S-TDM, we only allow one core to perform one access to the bus in a period.

Definition 1. A one-slot TDM schedule (1S-TDM) is a TDM schedule that has exactly one slot allocated to each core in every period.

3.3.3 Key observations

Although a 1S-TDM schedule guarantees a WCL bound for c_{ua} , we show that the WCL is proportional to the minimum of the cache capacity of c_{ua} and c_{ua} 's LLC partition size M , and cube of the number of cores. This results in a significantly large WCL. We illustrate this by making observations from two examples.

Consider the example in Figure 3.3 that has four cores, and a two-way set-associative LLC. The 1S-TDM schedule is $\{c_{ua}, c_2, c_3, c_4\}$. In ①, c_{ua} requests cache line X , which is not privately cached by c_{ua} . Hence, its L2 cache controller issues request for X at the beginning of $s_{c_{ua}}^t$ to the LLC. Cache line X maps to a cache set $set_{LLC}(X)$ in the LLC, and it also experiences a miss in the LLC. Since $set_{LLC}(X)$ has no empty lines, it must evict one cache line in $set_{LLC}(X)$. Suppose that $l_1 \in set_{LLC}(X)$ is selected for eviction. Note that l_1 is privately cached in c_3 denoted by $l_1 : c_3$. Hence, c_3 must also evict l_1 from its private caches in ②, resulting in a free entry in $set_{LLC}(X)$, denoted as $l_1 : -$. Clearly, the cache line to replace depends on the replacement policy. In this work, we assume a replacement policy that can select any of the cache lines. Even though we select l_1 in this

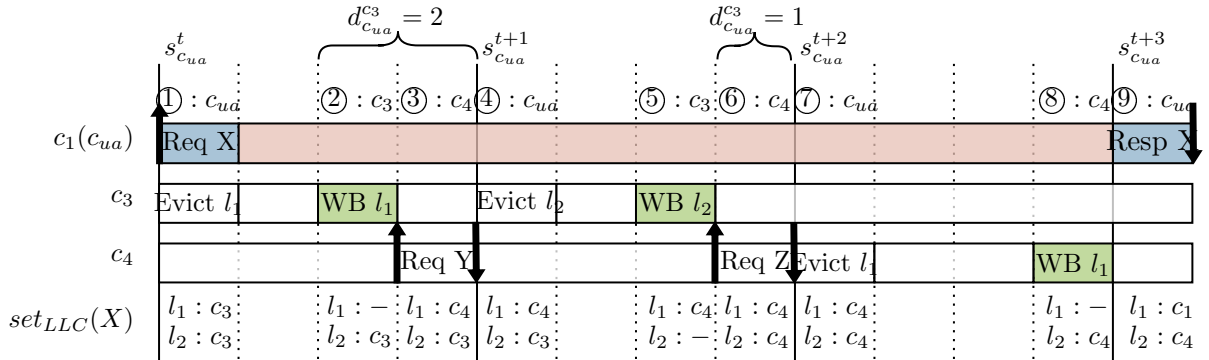


Figure 3.3: c_{ua} 's request to cache line X eventually completes.

example, our observation is agnostic of replacement policy. As a result, the observation applies to any replacement policy, including least-recently used (LRU).

Next, in ③, c_4 's L2 cache controller issues a request, and occupies the free entry, preventing c_{ua} from obtaining its response in slot $s_{c_{ua}}^{t+1}$. As a result, the LLC must evict a cache line in $set_{LLC}(X)$ to make space for c_{ua} , and the LLC evicts l_2 in slot $s_{c_{ua}}^{t+1}$ which is privately cached by c_3 . As before, c_3 evicts l_2 in ⑤, but it is occupied by c_4 in ⑥. Note that in the period starting at $s_{c_{ua}}^{t+2}$, this pattern of interfering c_{ua} from receiving its response cannot continue. When c_4 evicts l_1 in ⑧, there is no other core that can occupy the free entry; thus, c_{ua} will get its response in $s_{c_{ua}}^{t+3}$ at ⑨. This is guaranteed to occur because whenever any core other than c_{ua} occupies a free entry in $set_{LLC}(X)$, c_{ua} gets *closer* to being able to occupy a free entry in $set_{LLC}(X)$. For example, in ⑦, both cache lines are privately cached by c_4 , and any core making a request to $set_{LLC}(X)$ resulting in a miss requires c_4 to evict it from its private caches as well. Since, c_{ua} 's slot comes after c_4 's, it is guaranteed to occupy the free cache line entry due to c_4 's eviction. We characterize this *closer* effect by introducing a notion of distance (Definition 2).

Definition 2. For a 1S-TDM schedule S , the distance between two cores c_i and c_j , $d_{c_j}^{c_i}$, is the number of slots between the start of slot of c_i , and the start of c_j 's next slot.

Corollary 1. Given a 1S-TDM schedule S with N cores, the distance between any two cores c_i and c_j , $d_{c_j}^{c_i}$, $1 \leq d_{c_j}^{c_i} \leq N$.

Given a cache line l , we will use $d_{c_j}^{c(l)}(x)$ as a convenience to return the distance between the core that has privately cached l , and c_j at x . Using Figure 3.3, with a TDM schedule of $\{c_{ua}, c_2, c_3, c_4\}$, $d_{c_{ua}}^{c_3} = 2$ and $d_{c_{ua}}^{c_4} = 1$. With Definition 2, the example in Figure 3.3 can be interpreted in terms of distance: the core that caches l_1 changes from c_3 in slot ①, with a distance of $d_{c_{ua}}^{c_3} = 2$, to c_4 in slot ③, with a distance of $d_{c_{ua}}^{c_4} = 1$, and finally freed in slot ⑧. Similarly, the core that caches l_2 changes from c_3 in slot ① to c_4 in slot ⑥, and thus the distance of the core that caches l_2 decreases from 2 to 1. These examples highlight the following key observations.

Observation 1. Given a 1S-TDM schedule S , the distance of cache lines in $set_{LLC}(X)$ decrease when c_{ua} does not perform write-backs after issuing its request to cache line X and before receiving its response for X .

The decreasing distance articulates the effect of the core under analysis getting *closer* to occupying a freed cache line entry in $set_{LLC}(X)$. A direct consequence of observation 1 is that c_{ua} 's request will eventually complete as described in observation 2.

Observation 2. c_{ua} 's request will eventually complete.

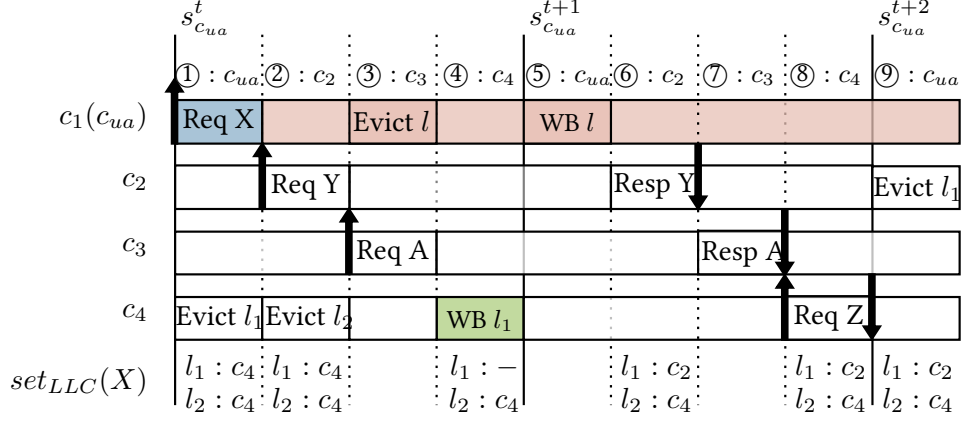


Figure 3.4: Distance of core caching l_1 increases after $s_{c_{ua}}^{t+1}$.

The main intuition behind this observation is that once the lines in $set_{LLC}(X)$ are privately cached by c_4 , a request for X by c_{ua} will succeed in the following period (Figure 3.3). This is because c_4 must evict the privately cached line due to inclusive property, which results in a free entry in $set_{LLC}(X)$ that c_{ua} can occupy. When $n \leq N$ cores share a partition with a 1S-TDM schedule and there are w ways in $set_{LLC}(X)$, for c_{ua} to occupy an entry in $set_{LLC}(X)$ in the worst-case, the distances of all w cache lines must experience the largest decrements. Since the maximal distance is n when c_{ua} caches a cache line, and the minimal distance is 1, c_{ua} must wait for the distances of all w cache lines to decrease from n to 1, accounting for $w(n - 1)$ decrements in the worst-case.

Note that we have not considered scenarios where c_{ua} performs write-backs before receiving the response for its request. When a core other than c_{ua} requests a cache line that is privately cached by c_{ua} , then c_{ua} would also need to perform write-backs due to inclusivity. The effect of write-backs on the distance is summarized in observation 3. After a write-back by c_{ua} , cache lines in $set_{LLC}(X)$ can be privately cached by a core with a larger distance compared to before c_{ua} performs the write-back.

Observation 3. Given a 1S-TDM, when c_{ua} performs a write-back after issuing its request to a cache line X and before receiving the response, the distances of cache lines in $set_{LLC}(X)$ increases.

Figure 3.4 shows a scenario with c_{ua} performing write-backs. There are four cores with 1S-TDM schedule of $\{c_{ua}, c_2, c_3, c_4\}$. Cache lines l_1 and l_2 are in $set_{LLC}(X)$, and are initially privately cached by c_4 . In ①, c_{ua} issues a request to cache line X that misses in the private caches and LLC. This is followed by c_2 issuing a request to cache line Y such

that $Y \in \text{set}_{LLC}(X)$ in ②, and it also misses in the private caches and LLC. The LLC selects to evict another line l_2 , which needs to be evicted by c_4 . In ③, c_3 issues a request to cache line A that causes c_1 to evict a cache line l . In ④, c_4 writes back l_1 freeing up an entry in $\text{set}_{LLC}(X)$. In the write-back slot of $s_{c_{ua}}^{t+1}$ ⑤, c_1 's request to X cannot be satisfied because c_1 has to perform an eviction. The free entry is thus occupied by c_2 . Note that the core that is caching l_1 has changed from c_4 to c_2 and thus the distance of the core caching l_1 increased from $d_{c_1}^{c_4} = 1$ to $d_{c_1}^{c_2} = 3$. In general, write-backs from c_{ua} allow a core with a larger distance to occupy a free entry in $\text{set}_{LLC}(X)$ that would have satisfied c_{ua} 's request. Therefore, the distance does not always decrease as in the case of Observation 1 when c_{ua} performs write-backs. We combine these two observations to develop a WCL analysis of a request. Henceforth, we assume the worst-case replacement policy: the LLC always replaces line with maximal distance for generality. Replacement policies such as LRU may tighten the bound but only by a constant factor.

3.3.4 WCL analysis for 1S-TDM schedule

We prove that the distances in $\text{set}_{LLC}(X)$ only decrease when c_{ua} does not perform a write-back in Corollary 2 (Observation 1). We bound the latency required for the distance to decrease. In Lemma 2, we show that the distance increases when c_{ua} writes back cache lines (Observation 2). Hence, when c_{ua} waits for its response, the distances in $\text{set}_{LLC}(X)$ show a pattern of decreasing and increasing. Finally, Theorem 2 combines Corollary 2 and Lemma 2 to express the WCL of c_{ua} 's request. We use our key observations to formulate an analysis to compute the WCL. Consider a multicore configuration with N cores interacting over the shared bus using a 1S-TDM schedule S , and n cores sharing a partition $\mathcal{P}$ in the LLC ($n \leq N$) with c_{ua} being one of the n cores. Throughout the analysis, we assume that c_{ua} 's request for cache line X misses in its private caches and the LLC, and $\text{set}_{LLC}(X)$ is full before c_{ua} 's request to cache line X is completed.

Lemma 1. *If c_{ua} 's request is not completed at slot $s_{c_{ua}}^{t+T}$, c_{ua} does not perform any write-backs, and $l_x \in \text{set}_{LLC}(X)$ is evicted in response to c_{ua} 's request in $s_{c_{ua}}^t$, then*

$$\forall l \in \text{set}_{LLC}(X) : d_{c_{ua}}^{c(l)}(s_{c_{ua}}^{t+T}) \leq d_{c_{ua}}^{c(l)}(s_{c_{ua}}^t).$$

Proof. We prove the lemma by contradiction and assume that $\exists l \in \text{set}_{LLC}(X) : d_{c_{ua}}^{c(l)}(s_{c_{ua}}^{t+T}) > d_{c_{ua}}^{c(l)}(s_{c_{ua}}^t)$. Then, before $s_{c_{ua}}^{t+T}$, there must exist two cores c_i and c_j , such that c_i frees the entry l and c_j occupies l after c_i frees l . The freeing-then-occupying by c_i and c_j increases the distance of l to be greater than $d_{c_{ua}}^{c(l_x)}(s_{c_{ua}}^t)$, that is, $d_{c_{ua}}^{c_i} \leq d_{c_{ua}}^{c(l_x)}(s_{c_{ua}}^t) < d_{c_{ua}}^{c_j}$. Assume

that c_i frees l in $s_{c_i}^q$ and c_j occupies l in $s_{c_j}^r$, then $s_{c_i}^q < s_{c_j}^r$. Furthermore, because $d_{c_{ua}}^{c_i} < d_{c_{ua}}^{c_j}$ and 1S-TDM is deployed, $s_{c_j}^r$ must be in the next period of $s_{c_i}^q$. There must be a slot of c_{ua} , $s_{c_{ua}}^p$, such that $s_{c_i}^q < s_{c_{ua}}^p < s_{c_j}^r$. Thus, there is a free entry in $set_{LLC}(X)$ in slot $d_{c_{ua}}^p$. Because c_{ua} 's request is not completed in slot $s_{c_{ua}}^{t+T}$, it is not completed in slot $d_{c_{ua}}^p$. The only reason that c_{ua} 's request is not completed in its slot when there is a free entry is that c_{ua} is performing a write-back, which contradicts the hypothesis that c_{ua} does not perform any write-backs. $\square$

Corollary 2. *If c_{ua} does not perform any write-backs, $l_x \in set_{LLC}(X)$ is evicted in response to c_{ua} 's request in $s_{c_{ua}}^t$ and c_{ua} 's request is not completed in $s_{c_{ua}}^{t+2(n-1)}$ then*

$$d_{c_{ua}}^{c(l_x)}(s_{c_{ua}}^{t+2(n-1)}) < d_{c_{ua}}^{c(l_x)}(s_{c_{ua}}^t)$$

Proof. Assume that at $s_{c_{ua}}^t$, cache line l_x is evicted, but it is privately cached by c_i such that $d_{c_{ua}}^{c_i} = d_{c_{ua}}^{c(l_x)}(s_{c_{ua}}^t)$. At a later slot for c_i , $s_{c_i}^q$ where l_x is written back, $q \leq t + 2(n - 2) + 1 < t + 2(n - 1)$. This is because there can be at most $(n - 1)$ pending write-backs in c_i 's PWB including the write-back for l_x . Before c_{ua} 's next slot, another core c_j must occupy l_x so that c_{ua} 's request is not completed. Due to 1S-TDM, if c_j occupies l_x , $d_{c_{ua}}^{c_j} < d_{c_{ua}}^{c_i} \leq d_{c_{ua}}^{c(l_x)}(s_{c_{ua}}^t)$. Applying Lemma 1, $s_{c_{ua}}^{t+2(n-1)} \leq d_{c_{ua}}^{c_j} < d_{c_{ua}}^{c(l_x)}(s_{c_{ua}}^t)$. $\square$

Lemma 2. *Given a slot $s_{c_{ua}}^t$ where c_{ua} performs write-back, then there exists an execution such that*

$$\forall l \in set_{LLC}(X) : d_{c_{ua}}^{c(l)}(s_{c_{ua}}^{t+1}) \geq d_{c_{ua}}^{c(l)}(s_{c_{ua}}^t).$$

Proof. In $s_{c_{ua}}^t$, let us assume that c_{ua} writes back a cache line as a response to an eviction caused by another core. Since c_{ua} is performing a write-back, it cannot issue a request; thus, its request cannot complete. Hence, for each of the free cache line entry l in $set_{LLC}(X)$ at $s_{c_{ua}}^t$, a core c_j can make a request to l after $s_{c_{ua}}^t$ which completes within one slot. Then, $d_{c_{ua}}^{c(l)}(s_{c_{ua}}^{t+1}) = d_{c_{ua}}^{c_j} > d_{c_{ua}}^{c(l)}(s_{c_{ua}}^t)$. For other cache lines $l' \in set_{LLC}(X)$ that are privately cached by other cores that are not evicted due to accesses made by some other cores, $d_{c_{ua}}^{c(l')}(s_{c_{ua}}^{t+1}) = d_{c_{ua}}^{c(l')}(s_{c_{ua}}^t)$ holds trivially. $\square$

Corollary 2 and Lemma 2 provide the cornerstone to derive the worst-case latency for c_{ua} in Theorem 1 and Theorem 2

Theorem 1 (critical instance). *Let $m = \min(m_{c_{ua}}, M)$, where $m_{c_{ua}}$ is the cache capacity of c_{ua} . The worst-case memory latency occurs when c_{ua} issues a request to line X such*

that: (1) before c_{ua} receives its response, all privately cached lines in c_{ua} are written back; (2) after each of these write-backs by c_{ua} , the distances of all cache lines in $set_{LLC}(X)$ increase to n ; (3) before each of these write-backs by c_{ua} , the distances of all cache lines in $set_{LLC}(X)$ decrease from n to 1; and, (4) after the last write-back by c_{ua} , the distances of all cache lines in $set_{LLC}(X)$ decrease from n to 1.

Proof. We prove the critical instance by construction. Suppose c_{ua} makes a request to X that maps to $set_{LLC}(X)$. Before c_{ua} receives its response, multiple write-backs from c_{ua} can occur in between as a result of other cores making requests. The maximal number of such write-backs, and the conditions under which they experience the maximum number of distance changes determines the critical instance. We have shown that the distances of lines in $set_{LLC}(X)$ only decrease between two write-backs while increases only when a write-back happens. The critical instance thus maximizes the number of distance changes between write-backs by means of *maximizing* the four parts: (1) The number of write-backs c_{ua} can perform in the worst-case as a result from other cores' requests. (2) The number of distance changes between any two write-backs by c_{ua} . (3) The number of distance changes before the first write-back of c_{ua} . (4) The number of distance changes after the last write-back of c_{ua} until it receives its response.

For (1), cores other than c_{ua} can force $m = \min(m_{c_{ua}}, M)$ write-backs for c_{ua} . This happens when m lines are privately cached by c_{ua} , and other cores make accesses that happen to map to the same sets as those m lines, which forces c_{ua} to write them back due to inclusion. This amounts to the maximal number of lines c_{ua} can cache in the partition $\mathcal{P}$. For (2), distance for a cache line can increase and decrease. Lemma 2 shows that after a write-back, $d_{c_{ua}}^{c(l)}$ is n in the worst-case for all $l \in set_{LLC}(X)$. Note that the distance cannot exceed n because this is the maximal number of cores that share $set_{LLC}(X)$ in $\mathcal{P}$. Lemma 1 ensures that $d_{c_{ua}}^{c(l)}$ strictly decreases and will not increase because of the absence of write-backs in between. In the worst-case, all $d_{c_{ua}}^{c(l)}$ decrease from n down to 1. For (3), in the worst-case, the distances of all w cache line entries in $set_{LLC}(X)$ are n when c_{ua} issues its request, and each distance decreases from n down 1 *before the first write-back* in the worst-case. Note that the distance cannot increase due to Lemma 1 and the absence of intermediate write-backs. Similar to (3), for (4), *after the last write-back*, the distances of all w cache line entries in $set_{LLC}(X)$ decrease from n down 1 before c_{ua} receives its response followed with one slot for c_{ua} to receive its response.

□

Theorem 2. Let $m = \min(m_{c_{ua}}, M)$, where $m_{c_{ua}}$ is the cache capacity of c_{ua} . The worst-case latency in number of slots of the request of the core under analysis c_{ua} , WCL , is given

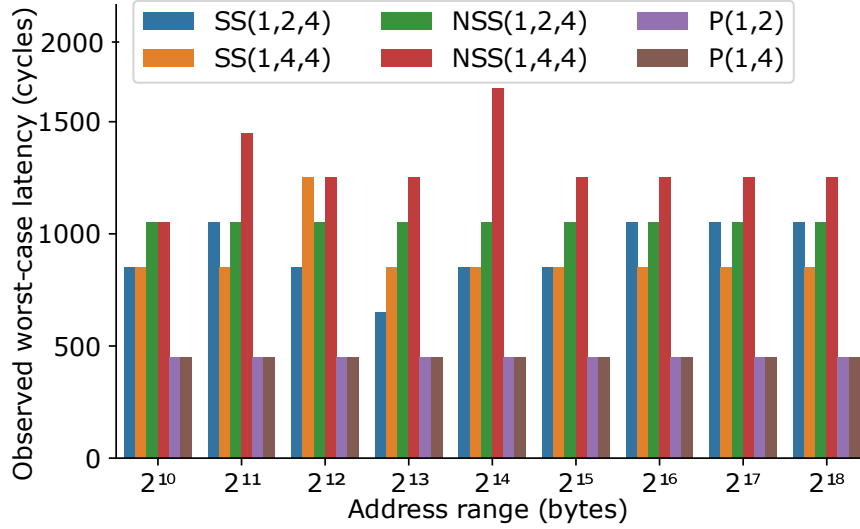
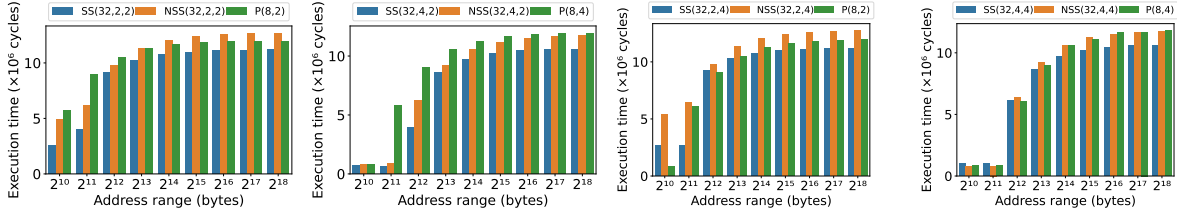


Figure 3.5: The observed WCL of SS, NSS and P.



(a) 2-core, 4096B partition. (b) 2-core, 8192B partition. (c) 4-core, 4096B partition. (d) 4-core, 8192B partition.

Figure 3.6: The execution time of synthetic workload with fixed total partition size.

by:

$$WCL = ((m + 1) \cdot A \cdot N + 1) \cdot SW, \quad (3.1)$$

where $A = 2(n - 1) \cdot w \cdot (n - 1)$.

Proof. Corollary 2 manifests $d_{c_{ua}}^{(l)}$ decreases by 1 every $2(n - 1)$ periods in the worst-case for $l \in \text{set}_{LLC}(X)$. Between two out of $(m - 1)$ write-backs from c_{ua} , all w cache lines' distances decrease from n to 1, taking $A = 2(n - 1) \cdot w(n - 1)$ periods. Before the first and after the last write-back, the distances of all w cache lines decrease from n to 1. c_{ua} receives its response in one slot. $\square$

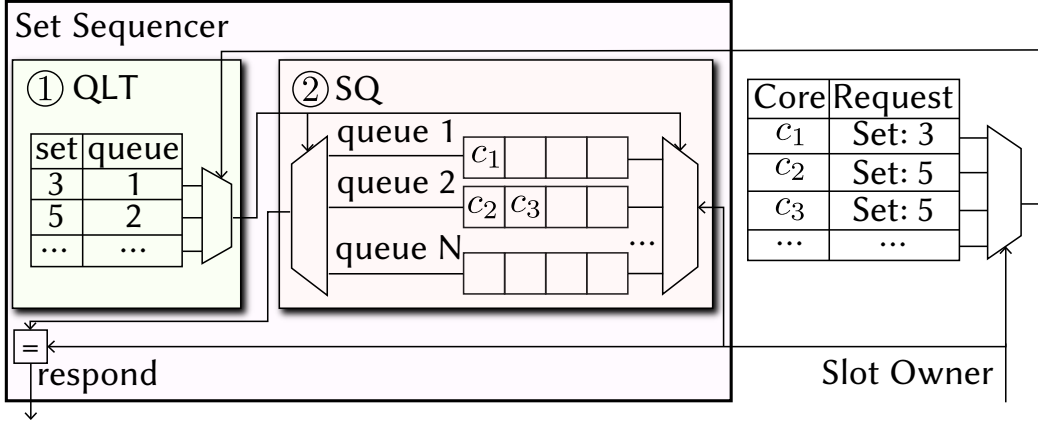


Figure 3.7: An illustration of set sequencer.

3.3.5 Set sequencer: Lowering the WCL

We propose a hardware extension called a set sequencer that significantly lowers the WCL. Recall that the WCL for a core under analysis c_{ua} is proportional to the minimum of either the cache capacity or c_{ua} 's LLC partition size, and cube of the number of cores. This is clearly a large bound. When using the set sequencer, the WCL bound is *independent* of the minimum of the c_{ua} 's cache capacity and c_{ua} 's LLC partition size M . We illustrate the main idea behind set sequencer using Figure 3.7, which contains two structures, a Queue Lookup Table (QLT) (①) and a Sequencer (SQ) (②). The set sequencer has two structures: a QLT and a SQ. The set sequencer contains one entry in the QLT for each set in the partition that has at least one pending LLC request. The entry maps the set to a queue in SQ. When there are multiple cores requesting a cache line from the same set, set sequencer stores the order in which the requests arrived at the LLC (broadcast order on the shared bus). Our key observations and the WCL analysis revealed that the distance increases whenever c_{ua} is prevented from occupying a free cache line entry due to another core intercepting it. By maintaining order using the set sequencer, we can guarantee that does not happen. Since the set sequencer only tracks outstanding memory requests for each core, QLT and SQ each contain up to N entries, respectively. Each QLT entry tracks the set index, the queue index, and each queue entry in SQ tracks the core index.

We present the WCL when using the set sequencer in Theorem 3.

Theorem 3. *The WCL of a request of the core under analysis c_{ua} when using the set sequencer, WCL_{ss} , is given by:*

$$WCL_{ss} = (2(n-1) \cdot n + 1) \cdot N \cdot SW. \quad (3.2)$$

Proof. In the worst-case, all other $(n - 1)$ cores issue their request before c_{ua} sends its request to cache line X to the LLC, and it is the last request in the set sequencer for a full set $set_{LLC}(X)$. For each request in the set sequencer, including c_{ua} , it takes $2(n - 1)$ slots for the core caching cache lines in $set_{LLC}(X)$ to write back a cache line and free an entry as a core performs $(n - 1)$ write-backs in the worst-case, and the evicted cache line is written-back last. Note that each such slot accounts for one period, which is $N \cdot SW$. Finally, c_{ua} requires one slot SW to receive its response, which accounts for another period. $\square$

3.4 Evaluation

Our empirical evaluation is performed with an in-house trace simulator that simulates the cache subsystem of a four-core system with the memory hierarchy as described in section 3.2. The L2 cache is a 4-way set-associative cache with 16 sets and the L3 cache is a 16-way set-associative cache with 32 sets that can be partitioned across the four cores. The cache line size is 64-byte.

Workload generation. We use synthetic workloads consisting of memory requests to random addresses within various address ranges. Each synthetic workload assumes the same address range for all cores. The memory address of each core is randomly generated such that they fall in the address range. We then apply offsets to the addresses of these memory requests to exclude data sharing between any two cores. For a certain address range, a core issues the same memory addresses across different partitioned configurations.

Notation. We use the following syntax to express partitioned configurations. (1) $SS(s, w, n)$: a partition shared among n cores with s sets and w ways with set sequencer, (2) $NSS(s, w, n)$: a partition shared among n cores with s sets and w ways and LLC services contending requests with best effort, and (3) $P(s, w)$: a partition with s sets and w ways that is uniquely occupied by a core. For $P(s, w)$, each core is assigned equally-sized partition.

3.4.1 Worst-case latency

Workload setup. To exercise the worst-case, we enforce a partition size of one set for all configurations. This forces as many conflicts as possible.

Results. Figure 3.5 confirms that the observed WCL of all configurations are within the analytical WCLs, which are 5000 cycles for SS , 979250 cycles for NSS , and 450 cycles for P . Although a distinct partition P yields the lowest WCL, recall that we wish to share partitions for cores whose real-time requirements are met with sharing. However, there might

be others that need distinct partitions P , which do indeed provide the lowest WCL. This is essential when the number of required functionalities deployed onto a single multicore increases. In the case of cores sharing a partition, the bound for SS can be employed. NSS shows a higher observed WCL compared to SS across all address ranges because distance can increase as mentioned in Observation 3.

3.4.2 Partition sharing and utilization

We evaluate the impact of partitioning when cores share a partition.

Workload setup. We use a 2-core and a 4-core setup with a fixed cache capacity that is partitioned. In SS and NSS , all cores share the same partition while in P , the fixed cache capacity is divided equally between all cores, and the set-associativity is fixed. Figure 3.6a shows that when the address range is 1024-byte or 2048-byte, the execution time is the same across SS , NSS and P . This is because the address range is less than or equal to the partition size.

When the address range exceeds the partition size, SS exhibits improved performance when compared to both NSS and P . In the 2-core setup with 4096-byte of partition size, SS achieves an average speedup of $1.34\times$ as is shown in Figure 3.6a. When the capacity is 8192-byte, SS achieves an average speed up of $2.13\times$ (Figure 3.6b). Such performance persists in the 4-core setup where SS features an average speedup of $1.10\times$ for 4096-byte partition size (Figure 3.6c) and an average speed up of $1.02\times$ for 8192-byte partition size (Figure 3.6d). The performance benefit when deploying set-sequencer is because the chain of back-invalidations is avoided, compared to when the set sequencer is not deployed.

3.5 Summary

We present an approach to predictably share LLC partitions. We use a constrained TDM policy with a set sequencer hardware extension, which results in a predictable and low WCL. Our empirical evaluation showed that we could reduce the WCL by 2048 times compared to a naive approach with the set sequencer.

Chapter 4

Eliminating back-invalidations for predicatble safety-critical multicores

Chapter 3 shows that the worst-case latency of accessing a shared last-level cache (LLC) partition can be unbounded, if not prohibitively large. The large worst-case latency occurs because the last-level cache must evict a cache line to make space for a new cache line. Depending on whether the cache line chosen for removal is present in the cores' private caches, this cache line removal from the LLC triggers additional invalidations of the cache line in the cores' private caches that have the cache line. Recall that we refer to this chain of invalidations as *back-invalidations* [167]. These back invalidations are a major contributor to the worst-case latency (WCL) of a memory request [129, 155]. While the technique in Chapter 3 addressed the limitation of LLC partitioning by guaranteeing a worst-case latency bound through the adoption of set sequencer, the worst-case latency is orders of magnitude larger than that with no LLC¹. While allowing cores to share LLC partitions is a welcomed step towards higher average-case performance, its high WCL questions its applicability in real-time platforms.

In this chapter, we aim at addressing the following questions: *Can we design a shared inclusive LLC with the following features: (1) all cores can access the full LLC (no partitioning constraints), (2) the WCL of a core's request is the same as that without an LLC, and (3) all cores can benefit the performance gains of LLC hits?* Such an LLC architecture is appealing for real-time multi-core platforms as it offers the performance benefits of using

¹Note that back-invalidations happen under certain situations based on the state of the LLC and private caches. However, the worst-case scenario for a core's memory request is when it triggers a back-invalidation in the LLC.

an LLC, but it does not incur additional latency to the WCL caused by back-invalidations. In other words, an LLC can be added to a multi-core platform at *no cost to the WCL*.

We answered the above question in the affirmative and presented a novel LLC architecture called ZeroCost-LLC (ZCLLC). ZCLLC borrowed the following key insight from a previous work on eliminating back-invalidations in the cache hierarchy of modern multi-core platforms to improve performance [27]: eliminating back-invalidations entails finding a location in the LLC to insert the requested cache line (that causes the back-invalidation) that is either vacant or has a cache line that is not present in any of the cores’ private caches. We progressively built towards a solution starting from this insight from ZIV [27] that eliminated back-invalidations on cores’ memory requests resulting in ZCLLC. A key contribution of ZCLLC was a mechanism that dynamically scheduled write-back of dirty cache lines from ZCLLC to main-memory. Such a mechanism ensured that a clean relocation entry was always available in ZCLLC, and as a result, eliminated back-invalidations on a cores’ memory requests that miss in ZCLLC. We further explore mechanisms to improve both the WCL and average-case performance after the elimination of back-invalidations. Specifically, ZCLLC used a work-conserving TDM arbiter with a fixed slot width as used in prior works [63, 75]. However, there are two downsides to this setup when used with an LLC. First, the arbiter fails to adapt to the short latency of LLC hits, hindering the performance gains from the LLC. Second, memory controllers are capable of handling multiple outstanding requests [148]. Since the TDM slot is typically large enough to complete a request, multiple outstanding requests are not permitted. Hence, the use of a TDM arbiter limits performance benefits and the possibility of further lowering the WCL. To this end, we analyze the downsides on performance and predictability with a fixed-slot TDM arbiter with ZCLLC, and systematically apply optimizations to mitigate these downsides. These optimizations lead to ZCLLC-OPT (ZCLLC with optimizations) that offers a lower WCL compared to ZCLLC as well as improved performance over ZCLLC.

The main contributions of this chapter are:

- We analyze the impact of back-invalidations on the WCL and describe the ingredients necessary to eliminate back-invalidations.
- We describe a novel LLC architecture, ZCLLC, that recognizes the need to combine ZIV with ROC to eliminate back-invalidations while bounding the WCL, and enforces a runtime invariant to ensure the existence of clean entries in the LLC to further lower the cost on the per-request WCL.
- We systematically analyze the impact on the WCL when deploying a work-conserving TDM arbiter when using ZCLLC, and propose optimizations resulting to an improved

LLC architecture, ZCLLC-OPT.

- We present a WCL analysis of ZCLLC-OPT and show that ZCLLC-OPT offers lower WCL for a memory request compared to ZCLLC.
- We evaluate ZCLLC-OPT in a detailed micro-architectural simulator and validate the worst-case bounds derived from our latency analysis. We also implement the state-of-the-art LLC partition sharing techniques and general-purpose configurations, with which we integrate our proposed optimizations. Our analysis reveals that with ZCLLC-OPT, the analytical WCL is 87.0%, 93.8%, and 97.1% lower than the state-of-the-art LLC partition sharing techniques for 2, 4, and 8 cores, respectively. In addition, our evaluation shows that ZCLLC-OPT offers speedups of $1.89\times$, $3.36\times$, and $6.24\times$ compared to the state-of-the-art LLC partition sharing techniques for 2, 4, and 8 cores setup, respectively.

4.1 System Model

4.1.1 Baseline Memory Hierarchy

Figure 4.1a shows our baseline system with N in-order cores and a two-level inclusive cache hierarchy. Each core has a private L1 instruction cache (L1I) and a private L1 data cache (L1D), and a private unified set-associative L2 cache (L2). Each core’s L1 caches connect to its L2 cache. All caches employ the least-recently-used (LRU) replacement policy. We further assume that there is at most one outstanding memory request per core. The cores’ L2 caches communicate with the main memory via two shared buses: one for commands and one for data. We use a work-conserving time-division multiplexing (TDM) arbitration [59,60] scheme that allocates one slot for each core to access the main memory. Under work-conserving TDM arbitration, if a core does not have a pending request to the shared LLC or main-memory at the start of its TDM slot, then the TDM slot is utilized by another core with a pending request to the shared LLC or main-memory. Our previous work [152] assumed that the TDM slot is long enough to complete one data transfer between the private L2 cache and the main memory. Such a deployment closely resembles platforms in [75,78,109,117] without a shared LLC. Recognizing the performance penalty of the assumption on TDM slot, in this work, we relax the constraint on TDM slot width, so that it is large enough to encompass the broadcast, the LLC access, and the corresponding main memory access. Note that, our approach can be extended to other TDM schedules.

The main memory controller accepts requests from the command bus and processes them in the order they are received. When a memory request misses in the L2, L2’s cache controller issues a request to fetch the data from the main memory. The main memory sends the requested data on the shared data bus. For brevity, we refer to the core issuing commands on the bus or receiving responses to be analogous to its L2 cache controller issuing commands and receiving responses. Caches transfer data in the granularity of a *cache line*, which we assume to be 64-byte wide. We use cache line A to represent the cache line indexed by address A . A predictable cache coherence mechanism such as the predictable Modified-Shared-Invalid (PMSI) cache coherence mechanism [75] facilitates coherent shared data communication between cores.

4.1.2 Memory Hierarchy with LLC

In Figure 4.1b, we augment the N -core baseline memory hierarchy with a *shared*, set-associative, write-back and write-allocate LLC. The cache hierarchy with the LLC is *inclusive*. This means that if a cache line is privately cached in L1 or L2, then this cache line is guaranteed to be cached in the LLC. If a cache line is not in the LLC then it is neither in the L1 nor L2. The LLC selects the LRU cache line (*victim* cache line) to replace on a conflict miss. The private L2 caches communicate with the shared LLC via the shared command and data buses. The LLC handles requests from the command bus in the order they are received and interacts with the main memory when a request misses in the LLC. We assume the capacity of the LLC to be greater than or equal to the aggregated capacity of all private caches [27]. An important component in our LLC is the tag array. Our tag array holds relocation information in addition to coherence-related states, and it is used for the same purpose as ZIV’s sparse directory. For ZCLLC, the tag array is set-associative and mimics the organization of the private caches. This is similar to [132]. We refer the reader to ZIV’s implementation for further details on the tag array’s design [26, 27].

Core memory operations. A core issues one of the following memory operations: Read, Write, or WriteBack. Read and Write correspond to loads and stores by the core. If a Read or Write hits in the private caches, the requested data is returned to the core. Otherwise, the Read or Write is forwarded to the LLC. The core issues WriteBack to relinquish its private copy of a cache line and *writes back* the data to the last-level cache. A WriteBack occurs when the core invalidates a cache line due to limited cache set conflict, or responds to another core’s request to cache a shared copy.

LLC Read/Write memory operations. We illustrate LLC Read and Write memory operations under our system model with an example, through which we introduce a

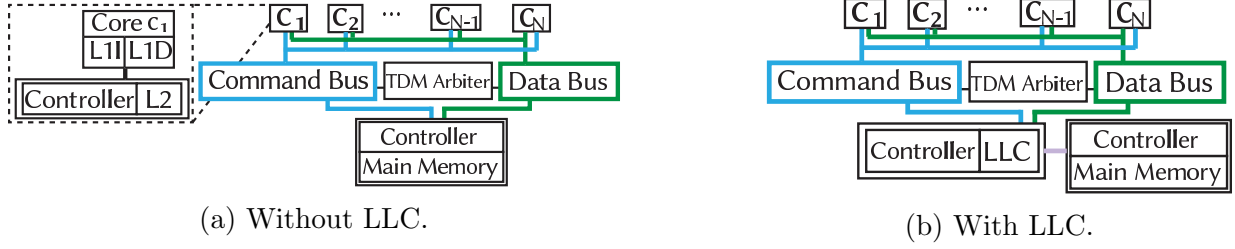


Figure 4.1: System model of ZCLLC.

well-known issue, *back-invalidation*, that may impact the WCL of memory requests when deploying an LLC. When the LLC receives a Read or Write request from a core, it fetches the requested cache line from the LLC or the main memory, and sends it to the requesting core's L2 cache controller. Consider the core under analysis, c_{ua} , issues a request to address A , denoted as $\text{Read}(A)$ or $\text{Write}(A)$. We denote the cache set that A maps to in the LLC as $s_L(A)$. Hence, c_{ua} 's request to cache line A maps to a cache set, $s_L(A)$, in the LLC. If A is cached in $s_L(A)$, then c_{ua} 's request is a *hit*, and the data of A is returned to c_{ua} . Otherwise, it is a *miss* and the LLC must fetch A from the main memory. Before fetching A , the LLC must ensure a vacant entry in $s_L(A)$ for A . If $s_L(A)$ has no vacant entries, then the LLC selects a victim line in $s_L(A)$, l , and evicts l to make space for A . However, victim l may be privately cached by another core. To ensure the inclusion property of the memory hierarchy, the LLC coordinates the eviction of l from all cores that have a private copy before invalidating the victim line in the LLC. We categorize back-invalidations into the following two types. (1) Given a back-invalidation from core c_i to core c_j , when $c_i \neq c_j$, the back-invalidation is called *cross back-invalidation*. (2) Otherwise when $c_i = c_j$, the back-invalidation is called *self back-invalidation*. Definition 3 defines the conditions under which a *back-invalidation* happens.

Definition 3. A back-invalidation from core c_i to core c_j happens when (1) c_i 's request to a cache line A misses in the private caches and the LLC, (2) the cache set corresponding to c_i 's requested line in the LLC, $s_L(A)$, is full, and (3) the victim selected by LLC for eviction is privately cached in c_j 's private cache.

Continuing with the example, the victim line l may be *dirty* (the data is modified when compared to the copy in the main memory); thus, it must be updated in the main memory. The LLC updates the main memory with the victim before servicing the request for A . For clean victims, the LLC only marks the entry in the LLC as invalid (no updates to the memory are generated).

LLC WriteBack memory operations. A WriteBack request for cache line A is denoted as $\text{WriteBack}(A)$. When the WriteBack is sent to the LLC, the LLC writes back the cache line received from the core to the LLC, or to the main memory. The WriteBack is the result of a self back-invalidation or a cross back-invalidation.

Note that the memory hierarchy design assumed in our system model does not feature eviction and fill buffers and miss-status-handling-registers (MSHRs) that facilitate memory-level-parallelism for improved average-case performance. This is because our system model is based on the models used in state-of-the-art predictable cache coherence mechanisms, which do not feature such buffers. As a first step towards accommodating such buffers, we explore the use of write-buffers in Section 4.6 that provide some overlapping of memory requests. We reserve the exploration of integrating these buffers into our model and using the insights from [141] to explore ZCLLC designs that offer memory-level-parallelism for future work.

4.2 Background

ZIV [27] presents a technique to eliminate back-invalidations in inclusive caches in general purpose systems. ZIV assumes that the capacity of the shared LLC is greater than the aggregated capacity of all private caches as stated in the system model. This assumption is essential for an LLC to be inclusive because the LLC must have enough capacity to guarantee that every privately cached line is also cached in the shared LLC. Thus, there *always* exist cache line entries in the LLC that are not privately cached by any private caches or vacant, which we refer to as *relocation entries* (REs). Note that an eviction of an RE does not incur any back-invalidation. The key insight in ZIV is that when the LLC selects a victim that is also privately cached, the LLC relocates the victim to an RE that is guaranteed to exist; thus, eliminating all back-invalidations. We use a register *nextRS* to track the location of an RE, and amend each LLC set with one additional bit, *HasRE*, to indicate the existence of at least one RE in that set.

We elaborate ZIV’s operation using the example in Figure 4.2a. A core, c_{ua} , requests for A and ZIV checks the tag array. There is no tag entry with A (①); thus, the request is a miss in the LLC. Note that the red markings indicate the state after the handling of the miss for A . Each entry in the tag array in ZIV tracks the tag, sharers, and a location tuple of set index and way index that points to the cache line the entry is monitoring in the data array. For example, cache line B is owned by core c_4 , and its data is stored in set 0 and way 0 in the data array. By changing the location tuple, ZIV allows a cache line to be relocated. The LLC queries the data array’s set 0 ($s_L(A)$) and observes that

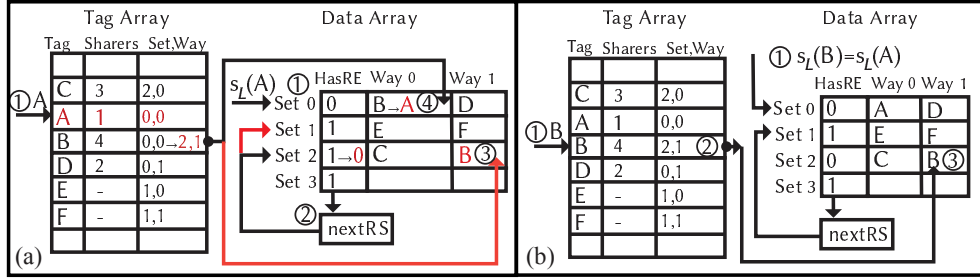


Figure 4.2: (a) ZIV operation in LLC to avoid back-invalidations. (b) ZIV operation in LLC on a hit to a relocated cache line.

there are no vacant entries since cache lines B and D occupy both ways. We assume that both lines are privately cached as well. ZIV selects B as the victim in $s_L(A)$. ② The LLC queries the *nextRS* register for the location of a RE for B 's target location. Suppose it points to set 2. ZIV uses an additional bit for each set in the data array, *HasRE*, to indicate a cache set with an RE. For example, set 0 does not contain an RE; thus, *HasRE* is 0. *nextRS* stores the set index of a cache set with an RE. Once a relocation happens, *nextRS* is updated to another cache set with RE in round-robin order, which is set 1. ③ Then, the LLC relocates B to the target location in set 2, and updates the location tuple of B 's tag array entry to set 2, way 1. Future requests to the LLC that access B use the set and way indices in the tag array to correctly locate the data array entry. ④ Finally, the LLC fetches A from the main memory and use the relocation entry to store A . The state of the LLC after relocating cache line B is shown in Figure 4.2b.

In ZIV, the access to a non-relocated cache line, such as A , follows the conventional data path, where the tag array and data array are queried in parallel, revealing that the access hits in the LLC. The access to a relocated cache line, such as B , involves an extra data array query, as we show in Figure 4.2b. Similar to access to a non-relocated cache line, the access to cache line B starts by querying the tag array and data array in parallel (①). The query to the tag array is a hit while the query to the data array is a miss. Recall from Figure 4.2a that cache line A and cache line B map to the same set ($s_L(A)$), and cache line B is relocated to set 2; hence, the miss in the data array. This indicates that cache line B was relocated in the data array. This means the LLC uses the location tuple from the array query to access the data array at set 2, way 1 (②). A second query to the data array will successfully retrieve the data (③). This second data array query lengthens the critical path to access a relocated cache line compared to non-relocated cache lines. However, ZIV [27] noted this to incur an additional latency of 3 cycles for extreme configurations such as a system with 768KB

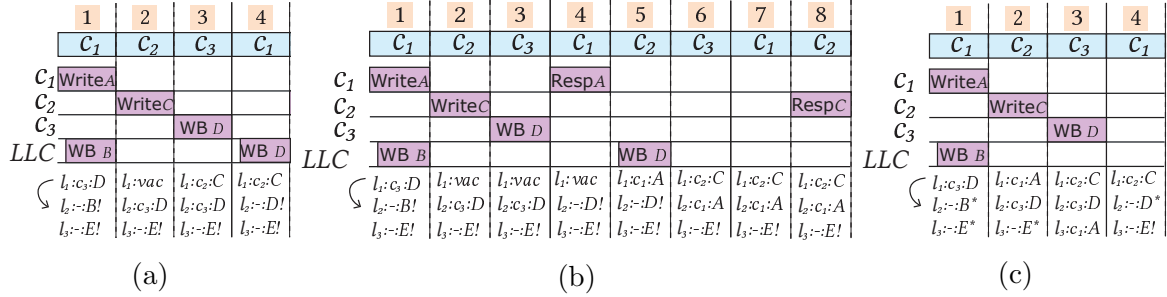


Figure 4.3: Different approaches of incorporating ZIV in an LLC: (a) employing ZIV by itself results in unbounded WCL (b) ZIV with ROC (c) Lower WCL with memory update.

aggregated L2 capacity and a 1MB LLC, where the aggregated L2 capacity is close to the LLC capacity. This is negligible compared to the WCL, which is dominated by the main memory access. To avoid the large overhead of combinational logic for determining *nextRS*, ZIV [27] employs a bit-masking technique to determine *nextRS* such that it is practical to compute *nextRS*. Specifically, when there are multiple cache sets with REs available, the *nextRS* is updated to the cache set according to round-robin. ZIV reported that it took 6% of a 1MB LLC storage for supporting cache line relocation. The storage scales linearly with the LLC capacity. We use the same technique as [27] to enable cache line relocation and to remove back-invalidations.

4.3 The Impact on WCL when Using a Shared LLC

Augmenting the memory hierarchy with a shared LLC for predictable multicore systems must be done carefully. This is because such a change in the memory hierarchy requires WCL analysis techniques to include the latency of accessing the LLC [81, 155, 164]. In this section, we show an alternative approach that incorporates a shared LLC with no partitioning constraints that has a lower WCL compared to state-of-the-art LLC partitioning approaches.

We use the asymptotic worst-case latency (AWCL) bound of a memory request [76] to present our insights and observations regarding state-of-the-art works on predictable LLC architectures and build ZCLLC, a novel predictable LLC architecture. AWCL is the WCL asymptotically in terms of the number of cores and is a succinct formulation to understand how the WCL of a memory request under a certain mechanism scales with the number of cores.

4.3.1 Expository Setup

We begin our exposition with the model in Figure 4.1a that has no LLC. Then, we explore the effect of adding an LLC into the memory hierarchy resulting in Figure 4.1b.

Setup without LLC as in Figure 4.1a. Simply for illustration, suppose we restrict the cores in Figure 4.1a to only make accesses to distinct cache lines. This disallows accesses to shared data. With this setup, every memory request can complete within a slot. This is because the length of the slot includes the latency for transferring the data between the private caches and the main memory. Hence, the WCL for a memory request occurs when a memory request just misses the start of its TDM slot, and must wait for its next slot to perform the access. As a result, a core must wait for a TDM period in the worst-case before it successfully issues its request. Thus, the memory request suffers the worst-case arbitration latency of $O(N)$ slots resulting in an AWCL that is linear in the number of cores (N). There are several recent works [63, 76] that have a similar setup as this one, but they also allow accesses to shared data with an AWCL of $O(N)$. ZCLLC applies to those works as well.

Setup with LLC as in Figure 4.1b. When we add an inclusive shared LLC to the setup, we end up with the system model shown in Figure 4.1b. This addition requires us to investigate whether the WCL of a memory request has changed. Specifically, adding an LLC introduces additional memory operations in the form of back-invalidations on LLC cache misses in order to make space in the cache hierarchy to accommodate the data returned from the main-memory for these cache misses. As a result, the WCL analysis must account for interference caused by these back-invalidations.

A recent work [155] showed that adding a shared inclusive LLC and enforcing an ordering constraint on which requests occupy vacant cache line entries yields an AWCL of $O(N^3)$. The $O(N^3)$ occurs because there are $O(N^2)$ back-invalidations in the worst-case that cause interference on a request. We denote this ordering constraint as *request ordering constraint* (ROC). ROC prevents a younger request from occupying a vacant entry in the LLC that was released for an older request. The key intuition behind the cubic bound with ROC is that in the worst-case, there are $O(N^2)$ self back-invalidation and cross back-invalidation that can be pending as a result of the ROC for a request, and each back-invalidation requires $O(N)$ slots to complete, which results in $O(N^3)$.

We highlight that by simply adding a shared inclusive LLC, the AWCL that was linear without the LLC ends up being cubic with the LLC. We find this to be a prohibitive cost to incur for adding a shared LLC. We acknowledge that there are other approaches that circumvent this cost. For example, a well practiced approach, LLC cache partitioning [46],

offers an $O(N)$ AWCL. However, this comes with caveats that disallow data sharing, and require kernel and OS changes [18, 29]. More importantly, all the difficulties related to LLC partitioning such as partition underutilization, and poor scalability to larger number of cores continue to persist. Our hope in this work is to present an approach where we can incorporate an inclusive shared LLC without incurring additional overhead to the WCL, and correspondingly to the AWCL. The significance of this approach is that WCL analyses do not need to account for interference as a result of adding an LLC. Hence, the original WCL bounds as computed without the LLC hold.

4.3.2 Observations and Insights

Question 1. *Given that back-invalidations are the main sources of interference when introducing an LLC, can we eliminate all back-invalidations?*

As explained in Section 4.2, the relocation strategy employed in ZIV can eliminate back-invalidations for multicore systems. However, adopting ZIV for predictable multicore systems must be carefully examined for its impact on the WCL.

Unbounded WCL when using ZIV by itself. We discovered that employing ZIV by itself with a TDM arbitration scheme established in prior works [75] results in an unbounded WCL. The TDM arbitration scheme enforces that one TDM slot is large enough to conduct one data transfer between the private cache and the main memory. As a result, the LLC may fulfill a request across multiple TDM slots. Such a TDM arbitration scheme allows the LLC to interleave and handle requests non-atomically, yielding improved average-case performance. However, the TDM arbitration scheme only specifies the order in which the cores access the LLC, and does not specify the order in which the LLC handles the requests. *This separation, if left under-specified, can result in unbounded WCL.* In Figure 4.3a, consider that core c_1 issues a request to A in slot 1 that maps to $s_L(A)$, which has one cache line entry l_1 in the LLC. The LLC caches three cache lines B , D , and E , where D is privately cached while B and E are dirty lines, but not privately cached by any core. Since, l_1 has valid data D that is privately cached by c_3 , $s_L(A)$ has no vacant entry. Since $s_L(A)$ has only one cache line, l_1 is selected as the victim line, and it must be relocated. Suppose that the target location selected by ZIV is l_2 , which has a dirty entry (denoted by !). Before relocating l_1 , the LLC must update the memory with contents of l_2 . Recall that the length of a slot only permits one access to the main memory; hence, this update to memory consumes c_1 's entire slot 1. At the end of slot 1, l_1 is vacant. However, in slot 2, c_2 makes a request to C that also maps to $s_L(A)$, and occupies this recently vacated entry l_1 . In slot 3, c_3 writes back D in 3 for its next request. Notice that c_2 usurped the vacant

entry l_1 that was freed for c_1 's request to A . Hence, in c_1 's next slot 4, c_1 has not completed its request to A ; thus, the LLC attempts to complete the request. However, $s_L(A)$ still has no vacant entries causing the above steps to repeat. This pattern can continue indefinitely resulting in the WCL being unbounded. We observe that the key reason for the unbounded scenario is the same for which ROC was proposed as a remedy [155]. Therefore, we combine the enforcement of ROC with ZIV, and examine its effect on the WCL.

ZIV with ROC. Figure 4.3b shows an illustration where we combine ZIV with ROC. We denote this configuration as ZIV+ROC. We observe that a WCL bound exists, and the AWCL is $O(N^2)$. We use the same memory requests in Figure 4.3a that resulted in the unbounded WCL in Figure 4.3b with ZIV+ROC. In slot 1, c_1 issues a request for A . The victim selected is l_1 . ZIV relocates l_1 to l_2 to avoid back-invalidation. Since l_2 is dirty, the LLC updates the main memory with B . These steps are exactly the same as in Figure 4.3a. In slot 2, c_2 issues a request to C . Unlike before, ROC prevents c_2 from occupying l_1 even if it is vacant. This is because l_1 was freed as a result of c_1 's request for A . In slot 3, c_3 updates the memory with D to make space for its next request. In slot 4, the LLC fetches A from the main memory, fills l_1 in the LLC, and sends A to c_1 . Similarly, the LLC attempts to fulfill c_2 's request in 5 and relocates l_1 to l_2 , updating the dirty cache line D in the memory. Finally, c_2 obtains the response and completes its request in 8.

Although ZIV eliminates back-invalidations, ROC can delay a request. This occurs when the target location is dirty, and must first be updated in the main memory. This consumes the entire slot as shown in slot 1 in Figure 4.3b. While waiting for the core's next slot, other cores can issue requests, and ROC queues them. Thus, in the worst-case $O(N)$ requests to the same cache set such as the one in 2 in Figure 4.3b could be queued, and each one may need to update the main memory. This yields $O(N^2)$ AWCL following Theorem 4 and Theorem 5.

Theorem 4 (Critical instance for ZIV-ROC). *The worst-case latency of c_{ua} 's memory request to a cache line that maps to $s_L(A)$ happens when (1) c_{ua} misses its slot, and has to issue its request in its request slot, (2) all $(N - 1)$ cores other than c_{ua} issue their requests to cache lines that map to $s_L(A)$ and c_{ua} is the last core issuing its request abiding ROC, and (3) each of the N requests causes relocation of cache line entries from $s_L(A)$ to a target that is dirty.*

Proof. In the worst-case, c_{ua} misses its slot. Its next slot is a write-back slot for WriteBack requests due to predictable arbitration between WriteBack and REQ requests. Hence, c_{ua} must for another period to issue its request to a line mapped to $s_L(A)$ in its next slot. While waiting, all other $(N - 1)$ cores issue their requests to the cache lines that also map

to $s_L(A)$ before c_{ua} issues its request. Due to ROC, the LLC handles c_{ua} 's request after these $(N - 1)$ requests. Note that this is the maximal number of requests that can be ordered before c_{ua} 's request because we allow at most one outstanding memory request per core. When processing these N requests, including c_{ua} 's request, no back-invalidation occurs due to ZIV. However, all requests can cause ZIV to relocate a cache line from $s_L(A)$ to a target location that is dirty. This requires the LLC to write back the dirty line. This is the longest latency one request can suffer because otherwise, the LLC can fetch data from the main memory and fulfill a request within the same slot. $\square$

Theorem 5 (WCL of ZIV-ROC). *The WCL of c_{ua} 's memory request is*

$$WCL_{ZIV-ROC} = 2 \times N \times SW + N^2 \times SW + SW.$$

Proof. In the worst-case, c_{ua} misses its slot, and has to wait for another two periods before its request can be issued. This is because of the predictable arbitration between WriteBack and REQ requests. In total, it takes a latency of $\underline{2 \times N \times SW}$ to broadcast c_{ua} 's request.

Theorem 4 shows that in the worst-case, c_{ua} 's request is the last one ordered among N requests. Each of the N requests requires two slots to finish: (1) the LLC writes back a dirty cache line, and (2) the LLC fetches the data from main memory and fulfill the request. The two slots are within two periods, causing a latency of $N \times SW$. As a result, it takes a latency of $\underline{N^2 \times SW}$ for the LLC to finish all N requests. Finally, it takes one slot for c_{ua} 's request to complete. Summing up these latencies yields the worst-case latency. $\square$

Insights 1. We derive two insights when using ZIV+ROC. (1) We have to assume the target location is dirty in the worst-case. This means the target location must be updated in the main memory before performing the relocation. (2) The AWCL is $O(N^2)$. Therefore, using ZIV+ROC to add a shared LLC to the setup without the LLC increases the AWCL by a factor of $O(N)$ when compared to the setup with no LLC.

Question 2. *When relocating a victim line in ZIV+ROC, what happens if we select a target location that is guaranteed to be clean or vacant?*

Suppose we devise an approach that guarantees that on a relocation, there always exists a target location that is clean or vacant. Then, an update to the main memory is not needed in that slot. Consequently, the request that required the relocation could fetch the requested cache line in the same slot. Consider the example in Figure 4.3c, which shows the same requests from Figure 4.3b. The main difference is that l_2 and l_3 are clean with respect to the main memory, labeled as $l_2 : - : B^*$ and $l_3 : - : E^*$. Definition 4 formally defines such entries.

Definition 4. A clean relocation entry (*CRE*) is a cache entry in the LLC that is a vacant entry or holds a clean cache line that is not privately cached.

A CRE allows relocation of cache lines without triggering an update to the main memory. c_1 's request for A can use the CRE l_2 on relocation. This allows the LLC to use the latency that was previously used for updating the dirty cache line in the main memory to instead fetch data from the main memory. Consequently, c_1 can receive its response in the same slot. Guaranteeing a CRE on a relocation results in an AWCL that is $O(N)$.

Insights 2. Our main insight is that if we ensure the target location on relocation is always a CRE, then we can complete the request within that slot.

ZCLLC, uses ZIV+ROC and also ensures that a CRE is available on a relocation. ZCLLC ensures that in the worst-case, the number of CREs is always sufficient for the LLC to complete requests from cores without incurring updates to the main memory and back-invalidations. A key technique in maintaining a sufficient number of CREs is that ZCLLC performs memory update when a core voluntarily relinquishes its copy of cached data in its private cache resulting in a CRE. We explain these details in section 4.4.

4.4 ZCLLC: Shared LLC at No Cost to WCL

The key novelty in ZCLLC is in being able to incorporate a shared inclusive LLC into the memory hierarchy without any cost to the WCL. ZCLLC does this by eliminating back-invalidations, and guarantees that no updates to the main memory are necessary on a relocation. Hence, ZCLLC ensures that there is always a CRE in the LLC when a relocation is done. ZCLLC enforces the *vacancy invariant* on a replacement in the LLC. Guaranteeing the vacancy invariant requires dynamically deciding whether to update the main memory or not. The main benefit is that when the vacancy invariant is satisfied, a request is guaranteed to finish within one slot. We explain how the vacancy invariant plays a role in ensuring the existence of a CRE in this section. Note that in our design, WriteBack of a clean cache line is not silent, and must be handled by the LLC. We collect the symbols used in this chapter in Table 4.1.

4.4.1 ZCLLC: Micro-architectural Extensions

ZCLLC reuses ZIV's approach to relocate cache lines to a new cache set and to access a relocated cache line. Hence, ZCLLC has the same storage overhead as ZIV for supporting cache relocation.

Table 4.1: Symbols and names used in proofs and analysis.

Symbol	Explanation	Symbol	Explanation
SW	Slot width	T	L2 capacity
M	LLC capacity	N	Number of cores
$\mathcal{Q}$	Dirty lines cached in LLC but not cached in any of the private caches	$s_L(A)$	LLC set cache line A maps to

Counter. ZCLLC introduces a counter in the LLC to track the number of dirty cache lines that are valid in the LLC, but not cached in any private caches. Let $\mathcal{Q}$ denote the set of addresses of cache lines in the LLC that are not privately cached and dirty. ZCLLC tracks $|\mathcal{Q}|$.

Tracking CREs. We extend ZIV to support the tracking of CREs. Recall from Section 4.2 that ZIV tracks the cache set that has REs by augmenting each LLC set with a HasRE bit. ZCLLC reuses this bit to indicate that a cache set has at least one CRE, enabling the use of ZIV’s hardware to relocate cache lines to CREs. Following ZIV [27]’s approach of tracking RE with a round-robin fashion, ZCLLC tracks CREs by augmenting each cache set in the LLC with one bit indicating whether there exists an entry that is clean and not privately cached. This bit evaluates to 1 when such an entry exists and 0 when there is no such entry. For a w -way set-associative LLC with n sets, this amounts to an n -bit vector. Any set with a bit value of one contains a CRE and contains at least one entry that can be used by ZCLLC for relocation. In ZCLLC, we select the set that contains CREs in a round-robin order to minimize hardware cost. On every request, only bit positions corresponding to two sets are updated in the worst-case: (1) the cache set of the original request, and (2) the cache set of the relocation target. This incurs minimal cost in maintaining the bit vector.

4.4.2 ZCLLC: Dynamic Memory Update Policy on Replacements

ZCLLC dynamically enables main memory updates on replacements (WriteBack requests) from cores that relinquish their copy of data from the L2 cache. Note that a *main memory update*, or in short, a *memory update*, specifically means that the LLC synchronizes the contents of a dirty cache line in the LLC that is not privately cached by any core to the main memory, creating a CRE. On receiving a WriteBack, the LLC checks an invariant based

on the counters to decide whether to update the main memory’s copy of the addressed cache line that is dirty in the LLC. We call this the vacancy invariant (Definition 5). If this invariant is not satisfied, the LLC must perform a memory update. This ensures that when a future request results in a victim line that is privately cached, there is always a CRE to relocate to without triggering an update to the main memory.

Definition 5 (VINV). *The vacancy invariant (VINV) is a predicate given by the following inequality:*

$$M - |\mathcal{Q}| \geq N \cdot T. \quad (4.1)$$

M is the capacity of the LLC, and T is the capacity of each of the private caches as shown in Table 4.1. The left hand side of Inequality (4.1) is the number of cache entries in the LLC that is vacant, whose cache line is clean, or privately cached. These cache lines can be used or evicted without requiring an update to the main memory (UpdateMemory). The right hand side of Inequality (4.1) is the total number of entries in the private caches. When Inequality (4.1) holds, the LLC can always find a target location to relocate to without having to perform an UpdateMemory. As a result, satisfying this invariant eliminates back-invalidations on LLC cache misses resulting in a tighter WCL under ZCLLC. Note that for an application that only performs Read operations, $\mathcal{Q}$ is always empty because there are no dirty cache lines, and VINV is trivially satisfied.

Note that CREs in ZCLLC are cache lines that are either vacant or clean but not cached in the private caches. While Inequality (4.1) can be expressed as the availability of these cache lines to eliminate back-invalidations, framing the VINV in terms of M and $\mathcal{Q}$ better guides our design of ZCLLC. Specifically, determining whether ZCLLC satisfies VINV on a memory access requires ZCLLC to track only $\mathcal{Q}$; M , N and T are system model constants. Under cache coherence mechanisms, data contents of dirty cache lines evicted from private caches are written back to the LLC. As a result, tracking $\mathcal{Q}$ is already available as part of the cache coherence mechanism and ZCLLC leverages this to compute VINV. On the other hand, a ZCLLC design that tracks vacant entries or cache lines that are clean but not privately cached will require extensions to the cache coherence mechanism and additional hardware state resulting in hardware overhead.

4.4.3 Implementing ZCLLC in the LLC Controller

The LLC controller implements ZCLLC. We present the key implementation details in Algorithm 1. Table 4.2 describes the routines used in the implementation. ZCLLC receives

Table 4.2: Description of routines used in ZCLLC operation.

Routine	Description
ReceiveRequest()	Snoop request from command bus
ExtractRequest(req)	Extract information about request req such as request type, address and issuing core
HasVacantLine($s_L(addr)$)	Returns true if cache set $s_L(addr)$ has a vacant entry; return false otherwise
HasCleanLine($s_L(addr)$)	Returns true if cache set $s_L(addr)$ has a clean entry; returns false otherwise
SelectVictim($s_L(addr)$)	Returns a cache line to evict from LLC cache set $s_L(addr)$. Victim is selected based on LRU state
Relocate($victim, target$)	Relocate evicted cache line to new location $target$ in ZIVLLC
SingleCachedCopy($addr$)	Returns true if there is a single copy of $addr$ across the private caches; returns false otherwise
VINVHoldTrue(Q)	Returns true if VINV holds true; returns false otherwise
UpdateMemory(V)	Write-back data contents of cache line V to main-memory and mark the cache line location in ZCLLC occupied by V as invalid
SelectRelocationCRE()	Return a CRE for relocation as described in [27]
FetchData(req, c_i)	Fetch data for req from c_i from ZCLLC or main-memory

a request from the command bus and extracts information about the request (Lines 2-3). When the request type is a Read or Write and if the request is a miss, ZCLLC checks whether the cache set $s_L(addr)$ has a vacant or clean cache line (Line 6). If there is no vacant entry and there is no clean entry in $s_L(addr)$, ZCLLC selects a victim line in the set and relocates the victim to a CRE (Lines 7- 9). After this relocation, ZCLLC completes request by fetching data from the LLC or from the main-memory (Line 11). Note that VINV guarantees that there is always a CRE in the LLC to relocate a privately cached victim without incurring a back-invalidation or memory update. As a result, Relocate($victim, target$) will always return a valid CRE. If the request is a LLC hit or the request is a LLC miss but $s_L(addr)$ has a vacant or clean entry, then no relocation is necessary, and ZCLLC completes the pending memory request to main-memory.

When the request is a WriteBack, the core's L2 is relinquishing a copy of the cache line. As a result, ZCLLC needs to handle WriteBack and update the main-memory when necessary to create a CRE (Line 18-19). Depending on whether the dirty cache line being written-back is cached in multiple private caches (line 14), ZCLLC adds the cache line to Q . This addition to Q may violate VINV, and hence, ZCLLC checks whether VINV

Algorithm 1: ZCLLC

```
State      :  $Q$ 
Initialization:  $Q = \emptyset$ .

1 Loop
2    $req \leftarrow \text{ReceiveRequest}();$ 
3    $\langle type, addr, c_i, data \rangle \leftarrow \text{ExtractRequest}(req);$ 
4   if  $type = \text{Write}$  or  $type = \text{Read}$  then
5     // The request is a Write or Read;
6     if  $\text{IsMiss}(addr)$  and not ( $\text{HasVacantLine}(s_L(addr))$  or  $\text{HasCleanLine}(s_L(addr))$ ) then
7        $victim \leftarrow \text{SelectVictim}(s_L(addr));$ 
8        $target \leftarrow \text{SelectRelocationCRE}();$ 
9        $\text{Relocate}(victim, target);$ 
10    end
11     $\text{FetchData}(req, c_i);$ 
12  else
13    // The request is a WriteBack caused by replacement in L2;
14    if  $\text{SingleCachedCopy}(addr)$  then
15       $Q \leftarrow Q \cup \{addr\};$ 
16    end
17    if not  $\text{VINVHoldTrue}(Q)$  then
18       $\text{UpdateMemory}(addr, data);$ 
19       $Q \leftarrow Q \setminus \{addr\};$ 
20    end
21  end
22 EndLoop
```

holds on Line 17 to decide whether a memory update is necessary to guarantee VINV to provide enough CREs. If VINV does not hold, there are insufficient CREs and ZCLLC must perform a memory update (lines 18 - 19) to create CREs. Performing a memory update requires first choosing a dirty cache line in Q to evict from ZCLLC and updating the main-memory with data contents of this cache line. In the current implementation, ZCLLC returns the dirty cache line that is being written-back to ZCLLC (line 2). We reserve the exploration of strategies in choosing cache lines in Q to write-back to memory to improve average-case performance for future work. Otherwise, there are sufficient CREs and no memory updates are required.

Note that `SingleCachedCopy` has different behaviors depending on the predictable cache coherence mechanism deployed. Under predictable cache coherence mechanisms such as PMSI and PMESI [75] and variants of PMSI and PMESI described in [76], there can be only one copy of a dirty cache line across all private caches. As a result, line 14 will always return true under such predictable cache coherence mechanisms. Under a predictable cache

	1	2	3	4	5	6	7	8	9	10
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1
<i>Prv\$</i>	$\bar{c}_1 \bar{c}_2 \bar{c}_3$	$A \bar{c}_1 \bar{c}_2 \bar{c}_3$	$A B \bar{c}_1 \bar{c}_2 \bar{c}_3$	$A B C \bar{c}_1 \bar{c}_2 \bar{c}_3$	$B C \bar{c}_1 \bar{c}_2 \bar{c}_3$	$C \bar{c}_1 \bar{c}_2 \bar{c}_3$	$\bar{c}_1 \bar{c}_2 \bar{c}_3$	$D \bar{c}_1 \bar{c}_2 \bar{c}_3$	$D E \bar{c}_1 \bar{c}_2 \bar{c}_3$	$D E F \bar{c}_1 \bar{c}_2 \bar{c}_3$
C_1	Write A			WB A			Write D			
C_2		Write B			WB B			Write E		
C_3			Write C			WB C			Write F	
ZIVLLC					WB B	WB C				
l_1 :	$l_1:c_1:A$	$l_1:c_2:B$	$l_1:c_3:C$	$l_1:c_3:C$	$l_1:c_3:C$	$l_1::C^*$	$l_1:c_1:D$	$l_1:c_2:E$	$l_1:c_3:F$	
l_2 :	$l_2:$	$l_2:c_1:A$	$l_2:c_1:A$	$l_2::A$	$l_2::A$	$l_2::A$	$l_2::A$	$l_2::A$	$l_2::A$	$l_2::A$
l_3 :	$l_3:$	$l_3:$	$l_3:c_2:B$	$l_3:c_2:B$	$l_3::B^*$	$l_3::B^*$	$l_3::B^*$	$l_3::B^*$	$l_3::B^*$	$l_3:c_2:E$
l_4 :	$l_4:$	$l_4:$	$l_4:$	$l_4:$	$l_4:$	$l_4:$	$l_4:$	$l_4:c_1:D$	$l_4:c_1:D$	

Figure 4.4: Illustrative example.

coherence mechanism such as PMOESI [77], there can be multiple copies of a cache line in the private caches with one copy marked as dirty; this dirty copy has the owned or O coherence state. As a result, SingleCachedCopy will return false on a dirty cache line write-back in the owned or O coherence state, and will not update Q . In our evaluation, we use the predictable cache coherence mechanisms described in [76] where SingleCachedCopy always returns true on a write-back.

4.4.4 Illustrative Example

We use Figure 4.4 to explain ZCLLC's operation with a focus on ZCLLC's ability to enforce VINV. The main takeaway from this illustrative example is to highlight how ZCLLC ensures sufficient CREs are available to prevent back-invalidations on LLC misses.

Example setup. This example has three cores following the TDM schedule of $[c_1, c_2, c_3]$. Each private cache contains only one cache line. Figure 4.4 shows the content of each private cache in the *Prv\$* row. We use an underscore ($_$) to represent a vacant entry in the private caches and LLC, and mark clean cache lines in the LLC with a '*'. Vacant cache lines and the clean cache lines in ZCLLC are CRE targets to prevent back-invalidations on LLC misses. The LLC has four sets and each set contains one cache line labeled as l_1, l_2, l_3 and l_4 , respectively ($M = 4$ and $T = 1$). All private caches and the LLC start out not caching any cache lines ($Q = \emptyset$). Without loss of generality, all memory requests generated by the cores in this example map to set l_1 .

Slots 1 to 3. In slot 1, c_1 issues a request to A that completes in the same slot because the vacant entry l_1 is available for use. In slot 2, c_2 issues a request to B , which maps to

l_1 . Following line 6 to 9 in Algorithm 1, ZCLLC relocates the contents of l_1 to the vacant cache line l_2 ; thus, no back-invalidation. Hence, c_2 's request completes in slot 2 with B occupying l_1 , and A being relocated to l_2 . Similarly, c_3 issues a request in slot 3 that relocates l_1 to l_3 .

Slots 4 to 9. In 4, c_1 wishes to request cache line D , but c_1 's private cache is full. Hence, c_1 issues a WriteBack to write back A to the LLC. At the start of 4, on receiving the WriteBack request (Line 1 in Algorithm 1), the LLC checks if VINV holds if A in the LLC is *not* written to the main memory (Line 17 in Algorithm 1). Suppose that A is not written to the main memory, at the end of the slot, A will no longer be privately cached. The VINV from Inequality (4.1) holds: $4 - 1 \geq 3$. Thus, the LLC does not need to write A back to the main memory. In slot 5, c_2 issues a request to E , requiring a replacement on B ; thus, generating a WriteBack to the LLC. Similarly, the LLC checks whether VINV holds if B is not written to the main memory. If B is *not* written to the main memory, at the end of 5 $\mathcal{Q} = \{A, B\}$. In this situation, VINV does not hold: $4 - 2 \geq 3$. Hence, the LLC updates the main memory with B (Lines 18-19 in Algorithm 1). This is essential because if B is not updated in the main memory, there will be 2 dirty cache lines, A and B in the LLC leaving l_4 as the only CRE. However, there are 2 vacant entries in the private caches of c_1 and c_2 , allowing them to make 2 requests. With B written back to the main memory, VINV holds. In slot 6, c_3 performs a replacement to make space for its memory request. Similar to 5, the LLC must write back C . Now, the LLC ends up with $\mathcal{Q} = \{A\}$ that still needs to be written back to the main memory, and $\{B, C\}$ are clean with respect to the main memory. In slot 7, c_1 issues its request to D that maps to l_1 . The LLC can safely evict the current cache line in l_1 because it is clean (denoted by the '*'); main-memory was updated with data contents of cache line C at the end of slot 6. Then, the LLC fetches D from the main memory and sends D to c_1 . c_1 's request finishes within 7 without any back-invalidation. Note that although set l_4 is vacant in slot 7, no relocation to l_4 happens as l_1 has a clean entry that can be evicted without causing back-invalidations. In 8, c_2 's request E , and the cache line in l_1 is neither vacant or clean. This causes a relocation of the cache line in l_1 (D). ZCLLC relocates D to l_4 , to vacate l_1 and fetches E from the main memory. Note that although ZCLLC could choose l_3 since it is clean, it chooses the vacant entry l_4 to maximize LLC utilization. In 9, c_3 's request to F maps to l_1 and the LLC relocates E to l_3 , which is a CRE with clean cache line. No back-invalidation occurs.

When handling a WriteBack, the LLC ensures VINV holds if it does not perform a memory update. If this invariant does not hold, for example, when the LLC does not update the main memory with B in slot 5, the LLC will end up with only two CREs, l_1 and l_4 , for relocation at the end of slot 6. Two CREs for this example cannot guarantee back-invalidation-free service for three requests in slots 7, 8 and 9.

4.4.5 Characterizing ZCLLC’s Write Policy

ZCLLC is, by definition [62], a write-back cache and not a write-through cache, because a Write to the LLC always updates content in the cache and not the main memory. Further, the main memory only gets updated when ZCLLC performs a WriteBack. To maintain VINV, we make ZCLLC behave like a write-through cache. This occurs when a WriteBack triggers an UpdateMemory to guarantee that a specific number of CREs are always available as dictated by VINV. For ZCLLC to act like a write-through cache, the LLC state must be as follows: cache lines in the LLC are either privately cached or not privately cached, but dirty. If there are enough CREs as specified by VINV, a WriteBack does not trigger any main memory updates, and ZCLLC acts the same as a write-back LLC. It is noteworthy that ZCLLC will not be stuck in the LLC state that forces write-through cache behaviors. This is because Read requests from cores and their corresponding WriteBack will eventually increase the number of CREs even if VINV holds. In this work, we do not include a quantitative comparison with write-through caches. This is because prior works on predictable cache coherence [63, 75, 76] assume write-back caches. Extending predictable cache coherence protocols with write-through caches requires non-trivial coherence protocol changes whose correctness needs careful investigation, which is beyond the scope of this work.

4.4.6 Hardware and timing overheads of ZCLLC

ZCLLC uses the same logic as described in [27] to manage relocation and lookup relocated entries. As mentioned in [27], this logic results in an additional 6% overhead compared and a timing overhead of 1-3 cycles compared to a traditional 1MB LLC design. We take into account the timing overhead, setting the TDM slot-width in our empirical evaluation.

An alternative LLC design that eliminates back-invalidations without incurring any relocation of cache lines to different sets is one where the number of cache sets in the LLC is equal to that in the private L2 cache, and the associativity is equal to $N \times (W + 1)$ where W is the number of ways in the private L2 caches. Such a LLC design ensures that there is one way per set that is either vacant or has a cache line that is not privately cached by any cores, and hence, eliminates back-invalidations on a core’s request that misses in the private caches and LLC. However, the hardware and timing overheads for such an LLC design scales with the associativity of the private L2 caches and number of cores. On the other hand, ZCLLC is independent of the number of cores in the system and associativity of the private L2 caches. To put this in perspective, our evaluation in Section 4.7 uses a 8-core multi-core system where the private L2 cache is configured with 4 ways

and 128 sets and the ZCLLC is configured with 8 ways and 2048 sets. For this system setup, the alternate LLC design will need to be configured with 40 ways ($8 \times (4 + 1)$) and 128 sets to eliminate back-invalidations. We observe that this alternate LLC design incurs $2.1\times$ and $4.5\times$ hardware and timing overheads compared to ZCLLC configuration; we use CACTI [14] to measure the hardware and timing overheads based on a 22nm technology. Hence, ZCLLC is an attractive LLC design that eliminates back-invalidations, and its implementation is independent of core count and private L2 cache configuration.

4.5 ZCLLC's Correctness

In this section, we show that ZCLLC maintains the VINV in the LLC in the following way. First, we show that servicing Read and Write requests does not violate the VINV in Lemma 3. Next, we show that ZCLLC also maintains VINV when completing WriteBack requests in Lemma 4. Theorem 6 combines the lemmas and shows that VINV holds in ZCLLC.

4.5.1 Ensuring Vacancy Invariant in ZCLLC

Lemma 3. *Let Q be the set of cache lines in ZCLLC that are dirty but not cached in any of the cores' private caches. Let $\text{Req}(A)$ be a memory request to address A from a core that accesses ZCLLC where $\text{Req} \in \{\text{Read}, \text{Write}\}$. Let Q' be the set of cache lines in ZCLLC after $\text{Req}(A)$ is satisfied. Then,*

$$\text{VINVHoldTrue}(Q) \Rightarrow \text{VINVHoldTrue}(Q'),$$

Proof. Assume $\text{VINVHoldTrue}(Q)$ is true, that is, Inequality (4.1) holds:

$$M - |Q| \geq N \cdot T \quad (4.2)$$

Let core c_k issue request Req . This request can be a miss or a hit in ZCLLC.

Case 1: LLC miss. When Req is a miss, $A \notin Q$. ZCLLC must fetch A from main memory and send A to c_k . This means that after Req is satisfied, A is present in c_k 's private cache and ZCLLC. Hence, $Q' = Q$; A is not added to Q as A is cached in at least one private cache. As a result, $\text{VINVHoldTrue}(Q')$ is satisfied and VINV holds.

Case 2: LLC hit. If Req is a hit in the LLC, there are three sub-cases.

Sub-case 1: Request is a ZCLLC hit and A is dirty ($A \in Q$): A is brought

from ZCLLC into the private cache of the requesting core due to inclusivity of the cache hierarchy. As a result, after $\text{Req}(A)$, $\mathcal{Q}' = \mathcal{Q} \setminus \{A\}$. Since $|\mathcal{Q}'| < |\mathcal{Q}|$, the left-hand side of Inequality 4.2 increases by 1. Thus, $\text{VINVHoldTrue}(\mathcal{Q}')$ is satisfied and VINV holds.

Sub-case 2: Request is a ZCLLC hit and A is clean but not cached in any private caches: In this case, A is not present in $\mathcal{Q}$ as it is not dirty. As a result, $\mathcal{Q}$ remains unchanged before and after $\text{Req}(A)$. Hence, $\mathcal{Q}' = \mathcal{Q}$, and as a result, $\text{VINVHoldTrue}(\mathcal{Q}')$ is satisfied and VINV holds.

Sub-case 3: Request is a ZCLLC hit and A is cached in at least one private cache: This case is similar to the previous sub-case; A is not present in $\mathcal{Q}$, and $\mathcal{Q}$ remains unchanged before and after $\text{Req}(A)$. Hence, $\mathcal{Q}' = \mathcal{Q}$, and as a result, $\text{VINVHoldTrue}(\mathcal{Q}')$ is satisfied and VINV holds.

□

Key takeaway. Lemma 3 shows that servicing Read and Write requests in ZCLLC does not violate the VINV. This means that a Read or Write request from a core that reaches ZCLLC does not result in back-invalidations.

Lemma 4. Let $\mathcal{Q}$ be the set of cache lines in ZCLLC that are dirty but not cached in any of the cores' private caches. Let $\text{Req}(A)$ be a memory request to address A from a core that accesses ZCLLC where $\text{Req} \in \{\text{WriteBack}\}$. Let $\mathcal{Q}'$ be the set of cache lines in ZCLLC after $\text{Req}(A)$ is satisfied. Then,

$$\text{VINVHoldTrue}(\mathcal{Q}) \Rightarrow \text{VINVHoldTrue}(\mathcal{Q}'),$$

Proof. For a WriteBack request to A , Algorithm 1 modifies $\mathcal{Q}$ in Lines 4-19. Hence, it suffices to show that VINV holds after performing lines 4-19. Suppose VINV holds before the execution of lines 4-19 in Algorithm 1; hence, $M - |\mathcal{Q}| \geq N \cdot T$. Depending on the number of copies of A in the private caches, there are two cases:

Case 1: More than one core has A in their private caches: If more than one core has A in their private caches, then the condition on line 14 is false, and A is not added to $\mathcal{Q}$. As a result, $\mathcal{Q}' = \mathcal{Q}$ after $\text{WriteBack}(A)$. Since $\mathcal{Q}' = \mathcal{Q}$ and $\text{VINVHoldTrue}(\mathcal{Q})$ holds, $\text{VINVHoldTrue}(\mathcal{Q}')$ also holds.

Case 2: Only one core has A in its private cache: In this case, the dirty write-back of A to ZCLLC results in updating $\mathcal{Q}$ to include A . As a result, $\mathcal{Q}' \leftarrow \mathcal{Q} \cup \{A\}$. ZCLLC checks whether VINV holds for $\mathcal{Q}'$ as shown on line 17 in Algorithm 1. If $\text{VINVHoldTrue}(\mathcal{Q}')$ does not hold, then A is selected for eviction and its data contents are written to memory, which results in a vacant entry in ZCLLC. Hence, at the end of line 18, there is a CRE entry

in ZCLLC. A is removed from $\mathcal{Q}'$, and this results in $\mathcal{Q}' = \mathcal{Q}$. Thus, $\text{VINVHoldTrue}(\mathcal{Q}')$ holds. □

Key takeaway. Lemma 4 shows that ZCLLC ensures VINV holds after servicing WriteBack requests. This means that any subsequent Read and Write requests from cores that miss in the private caches and LLC will not result in back-invalidations as there is always a CRE available.

Theorem 6 (invariant). *Given $\mathcal{Q}$, ZCLLC maintains the VINV.*

Proof. (By induction on requests.) **Base case.** Initially, the LLC is empty, that is $\mathcal{Q} = \emptyset$. Inequality (4.1) reduces to $M \geq N \cdot T$. This is an assumption required to guarantee the inclusion property for the LLC.

Induction hypothesis. The invariant holds for the first y requests.

Inductive step. We prove that the invariant holds for the $(y+1)$ -th request. The $(y+1)$ -th request is either a WriteBack, a Write, or a Read. Lemma 3 shows that VINV holds for a Write or a Read. Lemma 4 shows that VINV holds when WriteBack is performed on a cache line. The two lemmas cover all scenarios of a request processed by the LLC. As a result, VINV holds for the $(y+1)$ -th slot. □

Key takeaway. Theorem 6 combines Lemma 3 and Lemma 4 and shows that ZCLLC maintains VINV.

Corollary 3. *Using ZCLLC, in the worst case, the number of back-invalidations and main memory updates in the LLC that interfere with the request from the core under analysis is 0.*

Proof. As shown in Theorem 6, VINV always holds in the LLC. Hence, WriteBack always completes in one slot. Read and Write also complete in a single slot without causing a back-invalidation or a write-back. Therefore, no back-invalidation or write-back causes interference. □

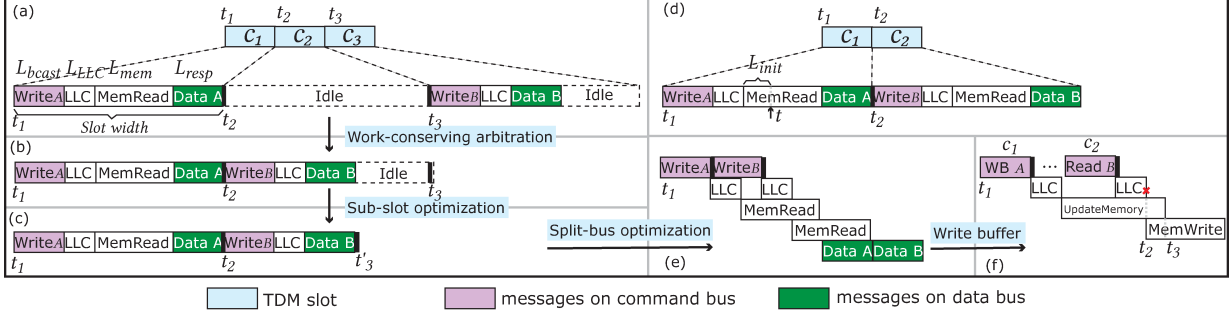


Figure 4.5: The examples show optimizations applied to ZCLLC, assuming a TDM schedule of $[c_1, c_2, c_3]$. (a) TDM arbitration used in ZCLLC introduces under utilization of the LLC. (b) Work-conserving arbitration removes slot level under-utilization. (c) Sub-slot optimization removes under-utilization occurred within a slot. (d) Main memory accesses are serialized. (e) Split-bus optimization removes under-utilization of the main memory. (f) Write buffer allows overlapping of UpdateMemory with other requests.

4.6 ZCLLC-OPT: Optimize ZCLLC for performance and predictability

In ZCLLC (ZCLLC-WC), a work-conserving TDM arbiter arbitrates which core gets to access the LLC in a given slot. Specifically, this involves arbitrating which core gets access to the command bus and data bus. The use of a work-conserving TDM arbiter contrasts with state-of-the-art LLC partitioning techniques [155], where a non-work-conserving TDM arbiter is used. For completeness of our discussion, we start by detailing the working mechanism of the non-work-conserving TDM arbiter if it is used in ZCLLC. Then, we highlight our observations of the inefficiencies of the mechanisms and propose our optimizations to overcome them. We also highlight that these optimizations are orthogonal to ZCLLC, and have also been applied to general purpose multicores. However, their application to a predictable LLC such as ZCLLC requires careful examination and analysis so that the WCL of memory requests is guaranteed. To this end, we also incorporate these optimizations with ZCLLC. We use ZCLLC-OPT to refer to *the ZCLLC that incorporates all the optimizations*.

4.6.1 TDM arbitration in ZCLLC

Prior state-of-the-art LLC partitioning techniques [75, 78, 155] use non-work-conserving TDM arbitration. Hence, we use non-work-conserving TDM arbitration as an example of showing the workings of the arbitration mechanism and the limitations of such an arbitration scheme. We refer to this configuration of ZCLLC as ZCLLC-NORMAL. ZCLLC-NORMAL follows prior works [75, 78] and its TDM arbiter grants access to a core following a TDM schedule. At the start of a slot, the arbiter grants access to *both* the command bus and data bus to a core according to the TDM schedule. Until the end of a slot width, the core can broadcast a command over the command bus, and send or receive data responses over the data bus. Figure 4.5a shows the timing of c_{ua} (c_1) sending a Write request that misses in the LLC. A request to address A in c_{ua} misses and requires it to broadcast the request over the command bus before t_0 . At t_1 , the TDM arbiter grants access to the command bus and c_{ua} 's request to A is broadcasted. The broadcast takes a latency of L_{bcast} to finish, after which, the LLC processes the request. This involves checking the tag array and data array, as described in Section 4.2 and takes a latency of L_{LLC} to finish. The LLC tag check shows that this request is a miss and hence it sends a memory read (MemRead) to the main memory. The MemRead takes L_{Mem} to finish and reaches the LLC, after which, the LLC sends the read cache line to c_{ua} over the data bus, taking a latency of L_{Resp} . To accommodate these operations, the slot width (SW) must be larger than the latencies of the operations combined. We next show that such an arbitration scheme incurs underutilization of the LLC, and we propose optimizations to overcome these underutilization.

4.6.2 Observations and solutions

Observations 1. *The LLC can be idle during a slot even if the LLC is capable of processing new requests.*

Figure 4.5a illustrates a scenario that demonstrates our observation. Between t_1 and t_2 , c_2 does not have a request to broadcast over the command bus. Meanwhile, c_3 has a request that misses in the L2 and needs to broadcast the request over the command bus. At t_2 , the arbiter allocates to c_2 a full slot, during which the LLC remains idle since it is only allowed to handle c_2 's request in the slot. At t_3 , the arbiter allocates a slot to c_3 and the LLC can now process its request to B .

Solution 1. To mitigate the presence of idle slots, exemplified by instances such as those in Observation 1, we employ work-conserving arbiters [59, 87]. In the event that a slot

scheduled for allocation would remain idle, the arbiter allocates the slot to the next core with a pending request according to the TDM schedule. Figure 4.5b shows the adoption of work-conserving arbiter. In Figure 4.5b, at time t_2 , if the slot is allocated to c_1 , it would remain idle. Consequently, the arbiter reallocates the slot to c_2 , which has a pending request. Using a work-conserving arbiter removes underutilization of the LLC between t_2 and t_3 . Note that such an arbiter will not alter the WCL because in the worst-case, a core still waits for a TDM period for its slot to be allocated.

Observations 2. *The arbiter provisions a slot width that accommodates main memory access even if the access to the LLC is a hit, negating the benefits of an LLC.*

Continuing the example in Figure 4.5b with the arbiter now being work-conserving, the slot allocated to c_3 from t_2 to t_3 allows the LLC to conduct a main memory transfer. However, since the request from c_3 hits in the LLC, the LLC can transfer the data back to c_3 over the data bus after it finishes checking the tag and data array and there is no need to access the main memory, leaving part of the slot idle and under-utilized.

Solution 2. To address the under-utilization within a slot, we propose a solution that dynamically varies the *allocated slot width* based on the tag check result of the LLC. In Figure 4.5c, instead of allocating a slot between t_2 and t_3 for c_3 's request, the arbiter allocates the slot between t_2 and t'_3 , reducing the slot width by the latency to access the main memory. This proposed modification improves the efficiency of the arbiter by dynamically adapting the slot width to the LLC hits and misses. By doing so, we anticipate a more effective utilization of resources without compromising the WCL of ZCLLC. Again, such an arbiter will not alter the WCL. Since in the worst-case, all memory accesses miss in the LLC, and the arbiter must allocate slots that are large enough to access the main memory. Note that Li et al. [94] proposed a variable TDM slot-based arbitration mechanism for DRAM memory controller transactions. On the other hand, our optimization dynamically varies the TDM slot based on LLC hits or misses.

Observations 3. *The LLC allows at most one outstanding main memory access.*

Solution 1 and Solution 2 enable cores to efficiently send requests to and receive memory responses from the LLC. However, main memory accesses sent by the LLC are still serial. In Figure 4.5d, c_1 and c_2 encounter memory requests to A and B , respectively before t_1 . Both requests miss in the L2 caches and the LLC, and hence the LLC must fetch the corresponding cache line from the main memory. The arbiter schedules c_1 's slot at t_1 , and c_2 's slot at t_2 . A memory controller allows multiple overlapping memory requests to improve performance, and hence it may accept new memory requests at t before the previous request finishes. The worst-case latency of the main memory accepting a new

request is L_{init} . As such, the main memory access corresponding to c_2 's request could start after t , if the arbiter is capable of scheduling c_2 's request earlier.

Solution 3. To address the underutilization of the main memory, we propose to decouple the arbitration of LLC access, main memory access, and the response bus from the TDM arbiter. With the decoupling, the TDM arbiter only arbitrates the access the command bus, where the accesses to the main memory can overlap. Additionally, the arbitration to access to the response bus follows the oldest-first (OF) order of their corresponding requests. Such a *split-bus* architecture decouples the access to the command bus and response bus, allowing memory requests to overlap. We acknowledge that the idea of a split-bus architecture has been proposed and adopted in prior works such as [63, 105], the impact of its use with an inclusive LLC specialized for safety-critical system is not present in prior works to the best of our knowledge. Figure 4.5e shows that the main memory access of c_1 's request and c_2 's request overlap, reducing the total latency of c_2 's request compared to Figure 4.5d.

Observations 4. Deploying a separate arbitration scheme in Solution 3 allows the overlap of MemReads. However, MemWrite caused by WriteBack cannot be overlapped. In Figure 4.5f, c_1 issues a WriteBack request to cache line A when the LLC has no CRE. This causes the LLC to perform a UpdateMemory to free up a CRE to maintain VINV. The UpdateMemory completes at t_3 and creates a CRE. Another core c_2 issues a request to cache line B before t_2 and misses in the LLC and no CRE is available at t_2 , when the LLC finishes the tag check. Since a CRE is created at t_3 , c_2 's request to B must be delayed until t_3 . We observe that such a delay is a result of the LLC creating CRE after UpdateMemory, and can enlarge the WCL.

Solution 4. We introduce a write buffer that buffers *cache lines* that undergo UpdateMemory. Recall that such an UpdateMemory is triggered in a WriteBack request (Line 18 in Algorithm 1). This write buffer hides the latency of creating a CRE through UpdateMemory. In Figure 4.5f, when the LLC performs UpdateMemory to A , the LLC moves the cache line A to the write buffer, creating a CRE immediately. This allows the MemRead in c_2 's request to start at t_2 instead of t_3 . Note that the data fill of c_2 's request into ZCLLC does not wait for the write-back of A to complete. Figure 4.6 shows the implementation of ZCLLC with write buffer. Entries of write buffer are composed of (1) the address (tag) of the cache line being written back, (2) the coherence state of the cache line, and (3) the actual data value of the cache line. We highlight the changes in operation compared to ZCLLC without write buffer. First, when the LLC performs the tag check, the write buffer is also checked (①). When the tag of an entry matches the requested cache line, it is considered an LLC hit. Second, the tag and data of a cache line may migrate between

the tag and data array, and the write buffer (②). Specifically, when the LLC performs an UpdateMemory on a cache line A , the tag and data of A are migrated to the write buffer. When a request hits in the write buffer, the tag, coherence state, and the corresponding data value are migrated from the write buffer to the tag and data array. Finally, when a request hits in the write buffer, the data value can be sent from the write buffer to the response bus (③). The size of the write buffer can be derived from the rate that an entry enters the write buffer, the rate that a request is sent to the main memory, and the latency that the main memory processes the request. Specifically, a request enters the write buffer every L_{bcast} cycles in the worst-case, and a request is sent to the main memory every L_{init} cycles. The worst-case latency that the main memory processes a memory request is L_{Mem} . Hence, the write buffer is occupied by a request by $(L_{\text{init}} + L_{\text{Mem}})$ cycles in the worst-case and the write buffer should buffer all requests that arriving during this latency, which is $\left\lceil \frac{L_{\text{init}} + L_{\text{Mem}}}{L_{\text{bcast}}} \right\rceil$. Note that in our setup, $L_{\text{init}} < L_{\text{bcast}}$ holds, hence the write buffer drains faster than it is filled; and it is possible for the write buffer to buffer all incoming requests without causing an overflow. In our empirical evaluation, we do not observe an overflow of the write buffer. As a result, the introduction of a write-buffer does not introduce any performance penalties.

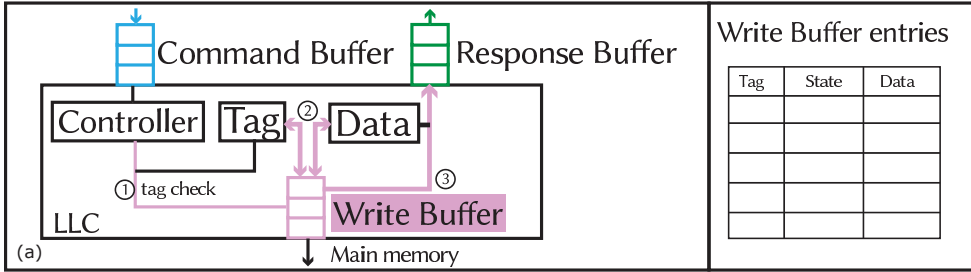


Figure 4.6: Write buffer implementation in ZCLLC.

Comparison: ZIV + ROC and ZCLLC with proposed optimizations. As described in Section 4.4, the LLC in ZIV + ROC may not find a CRE when processing a request from the core; hence, dirty cache line replacement can happen and occupy one slot. Such a difference persists even if all the optimizations are deployed in ZIV + ROC, resulting in a larger WCL compared to ZCLLC.

4.6.3 Worst-case Latency Analysis for ZCLLC-OPT

Our latency analysis for ZCLLC-OPT focuses on (1) the worst-case latency required to access the LLC, (2) the worst-case latency required for the data value to be available, and

(3) the worst-case latency for the data to arrive from the LLC to the L2 cache. We use L_{ArbOpt} to denote the worst-case latency required for a core to be serviced a slot on the command bus. A slot on the command bus is large enough to broadcast a Read, Write or a WriteBack. The analysis closely follows that of ZCLLC's that assumes a TDM arbitration scheme whose schedule is deployed with one slot per core; hence, $L_{\text{ArbOpt}} = (N - 1) \times L_{\text{bcast}}$. We refer the reader to [27] for the WCL analysis of a memory request under the ZCLLC design without the aforementioned optimizations.

Theorem 7. *For ZCLLC-OPT, the WCL of performing a write-back to the LLC or main-memory as a consequence of c_{ua} 's request in the LLC, WCL_{WTLM} , is*

$$WCL_{\text{WTLM}} = L_{\text{ArbOpt}} + L_{\text{bcast}}.$$

Proof. The WCL_{WTLM} is composed of the arbitration latency L_{ArbOpt} for c_{ua} 's request to be issued and the latency to perform the write-back to the LLC or the main-memory. Since the write-back from LLC to main-memory involves migrating the cache line from the tag and data array to the write buffer, it completes within one slot, which takes L_{bcast} . $\square$

Theorem 8. *For ZCLLC-OPT, the WCL for c_{ua} to receive a data response for its read or write request from the LLC when the response is ready, is*

$$WCL_{\text{Resp}} = (N - 1) \times L_{\text{Resp}} + L_{\text{Resp}}.$$

Proof. The arbitration of response follows the oldest-first work-conserving order. Hence, in the worst-case, $(N - 1)$ requests issued over the command bus are sent before the c_{ua} 's request that induces the response. Hence, $(N - 1)$ requests are issued before the response for c_{ua} is sent. Finally, the response of c_{ua} requires L_{Resp} to be sent over the response bus. $\square$

Theorem 9. *For ZCLLC-OPT, the WCL for the LLC to handle c_{ua} 's read or write request is*

$$WCL_{\text{LLC}} = L_{\text{LLC}} + L_{\text{Mem}}.$$

Proof. In the worst-case, the request misses in the LLC and requires accessing the main memory to fetch the requested data. The LLC takes L_{LLC} to handle internal bookkeeping, such as tag and data array query, and write buffer management. Recall from Observation 4 that we assumed that L_{init} is less than L_{bcast} according to our setup, hence the request to the main memory can be immediately accepted by the main memory once it enters the write buffer, without any blockage. $\square$

Theorem 10. For ZCLLC-OPT, the WCL of performing a fetch data from the LLC or main-memory as a consequence of c_{ua} 's request, WCL_{FFLM} , is

$$WCL_{FFLM} = L_{ArbOpt} + WCL_{LLC} + WCL_{Resp}.$$

Proof. This follows Theorem 8 and Theorem 9. □

Table 4.3 contrasts ZCLLC with other approaches to limiting the number of back-invalidations and the number of memory updates in the worst-case, together with the resulting AWCL. ZCLLC is the only approach that achieves no back-invalidations and no memory updates.

Table 4.3: Worst-case latency of various configurations. The worst-case latency accounts for the latency of both FFLM and WTLM.

Setups.	# of back-invalidations	# of mem. update	AWCL	WCL
ROC [155]	$(N-1)(N-1)$	$N-1$	$O(N^3)$	$((2 \times (N-1) \times N + 1)) \times N \times SW$
ZIV+ROC [152]	0	$N-1$	$O(N^2)$	$2 \times N \times SW + N^2 \times SW + SW$
ZCLLC [152]	0	0	$O(N)$	$2 \times N \times SW + SW$
ZCLLC-OPT	0	0	$O(N)$	$2 \times N \times L_{\text{bcast}} + N \times L_{\text{Resp}} + L_{\text{Mem}}$

4.7 Empirical Evaluation

We integrate our proposed mechanism in the gem5 [21] micro-architectural simulator. We simulate configurations with 2, 4, and 8 cores for ROC, ZIV+ROC, and ZCLLC. In addition, we implement a general-purpose cache-coherent data-sharing mechanism with no guarantees on timing predictability (GP). We also show the impact of the optimization techniques presented in Section 4.6 on the performance for each of the timing predictability mechanism. Specifically, we incrementally apply optimizations to each of the predictability mechanism, and their general purpose counterpart, GP. The baseline (*NORMAL*)

Table 4.4: Observed WCL (Obs.) and Analytical WCL (Analyt.) for different configurations (in cycles).

Core	ZCLLC-OPT		ZCLLC-NORMAL		ZIV+ROC-NORMAL		ROC-NORMAL		GP
	Obs.	Analyt.	Obs.	Analyt.	Obs.	Analyt.	Obs.	Analyt.	Obs.
2	138	150	612	640	1124	1152	1126	1143	1114
4	176	200	1124	1152	3172	3200	3095	3175	3316
8	249	300	2148	2176	9314	10368	9254	10287	24166

corresponds to configurations with a non-work-conserving TDM arbitration, such as the one [75, 155] used in prior works. By incrementally applying Solution 1 and Solution 2 over the baseline configuration, we obtain two configurations, *WC* and *SUBSLOT*, respectively. Finally, with Solution 3 and Solution 4, we obtain the configuration *OPT*. The name *OPT* follows from the fact that we apply all optimizations in such a configuration. To facilitate our discussion, we systematically denote a configuration by its mechanism to maintain predictability and the optimizations applied to it. Specifically, predictability mechanisms is one of ZCLLC, ZIV+ROC, ROC, and GP (GP meaning integrating an LLC naively). The optimizations technique applied to each predictability mechanisms is one of NORMAL, WC, SUBSLOT, and OPT. For example, ZCLLC with all the optimizations is named ZCLLC-OPT. We assume that each core is equipped with 4kB L1 private instruction and data caches, and 16kB private L2 caches. The private caches are backed by a shared 1MB LLC. The private caches and the shared LLC communicate through a shared bus deploying TDM arbitration with a slot width of large enough to allow one data transfer between the L2 and the main memory. For the ZIV+ROC and ZCLLC configurations, we use a TDM slot width of 128 cycles and for the configurations that do not feature the relocation logic (ROC and GP), we set the slot width to 127 cycles; we assume a timing overhead of 1 cycle due to additional lookup and management for relocated cache entries. Recall that ZCLLC-OPT uses the split-bus architecture optimization that decouples requests and responses. Hence, the TDM slot width for ZCLLC-OPT configuration is set to 8 cycles, which is large enough to broadcast a memory request on the command bus. We assume that $L_{\text{bcast}} = 8$ cycles, $L_{\text{LLC}} = 4$ cycles [27], $L_{\text{Resp}} = 9$ cycles, and $L_{\text{init}} = 4$ cycles. In addition, for OPT configurations, the slot width is $\max(L_{\text{bcast}}, L_{\text{LLC}}, L_{\text{init}}) = L_{\text{bcast}} = 8$ cycles and is large enough to encompass the broadcast, the LLC access, and the corresponding main memory access. The selection of these values reflects the number of message buffers that must be passed through between the cache controllers and the buses. We simulate the main memory with a pipelined main memory model with an access latency (L_{Mem}) of 100 cycles.

Our evaluation includes synthetic benchmarks to exercise the WCL and the SPLASH-3 [123] multi-core workloads. Our source code is publicly available at [150].

4.7.1 Worst-case Latency

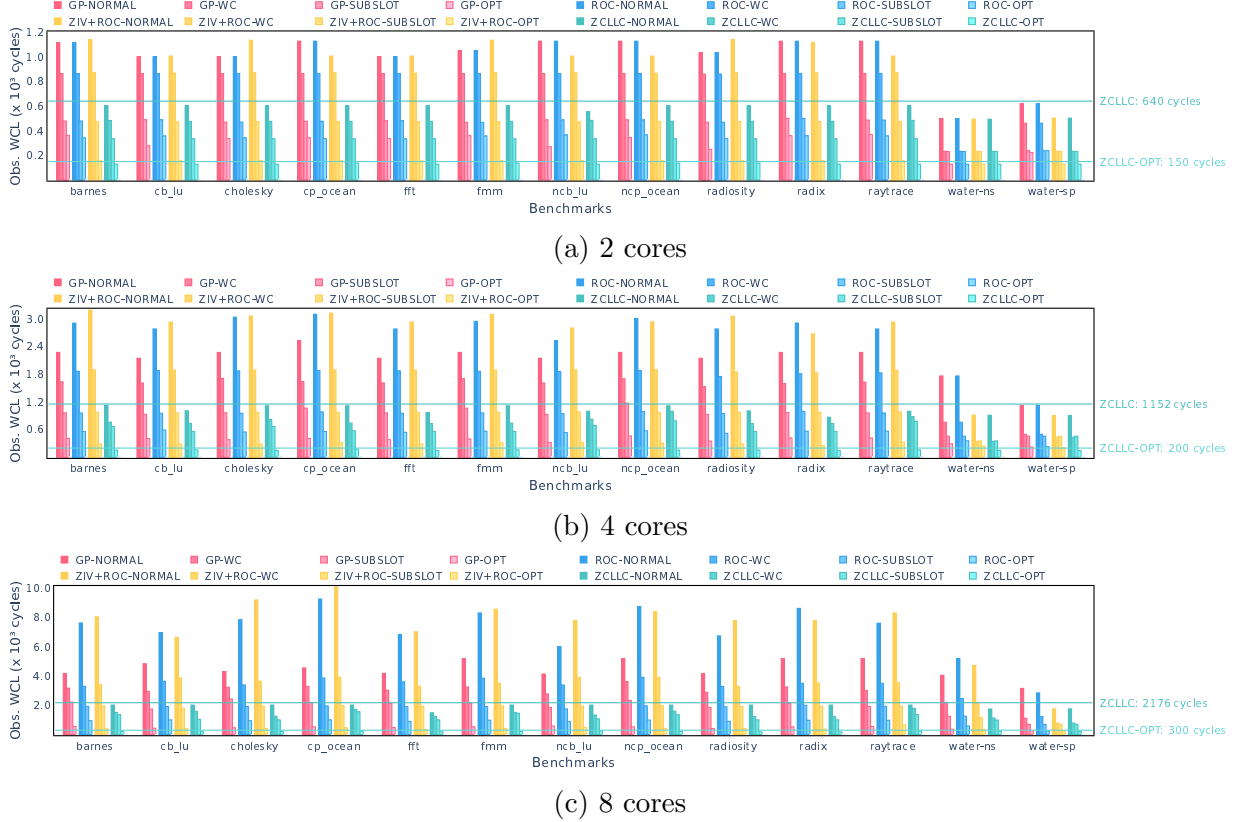


Figure 4.7: The observed worst-case latencies are shown for the following configurations: GP, ROC, ZIV+ROC, and ZCLLC for 2, 4, and 8 cores when running SPLASH-3 benchmarks. Optimizations to these configurations are applied incrementally to all configurations in the order of NORMAL (no optimization), WC, SUBSLOT, and OPT. The observed worst-case latencies are in cycles.

We begin by investigating the WCL properties of different configurations.

Synthetic benchmarks. We develop synthetic benchmarks in an attempt to exercise the WCLs for different configurations. Our synthetic benchmarks involve the cores ran-

domly generating memory accesses to locations that map to *the same cache set* in the LLC. Table 4.4 shows that for all configurations, the analytical WCLs safely bound the observed WCLs for all configurations with mechanisms to ensure timing predictability. We also show the observed WCLs for the GP configuration. The GP-NORMAL configuration exhibits a WCL of 840, 3566, and 18824 cycles, for 2, 4, and 8 cores, respectively. Note that the observed WCLs of 2-core, 4-core, and 8-core GP-NORMAL configurations exceed the analytical WCLs of the 2-core, 4-core, and 8-core ZCLLC-NORMAL configurations, respectively. The observed WCLs of the 4-core and the 8-core GP-NORMAL exceed the analytical WCLs of ZCLLC-NORMAL, ZIV+ROC-NORMAL, and ROC-NORMAL. Among the configurations, ZCLLC-NORMAL shows lower analytical WCLs of 640, 1152, and 2176 cycles compared to ROC-NORMAL and ZIV+ROC-NORMAL, and ZCLLC-OPT shows the lowest analytical WCLs of 150, 200, and 300 cycles. All analytical WCLs are calculated according to the WCL listed in Table 4.3.

SPLASH-3. We also evaluate the observed WCL with the SPLASH-3 benchmarks. SPLASH-3 benchmarks contain a wide variety of complex parallel multi-threaded applications that exercise data sharing and synchronization across cores. We use SPLASH-3 benchmarks to study the impact of these inter-core interactions on performance and WCL when a system deploys ZCLLC and the optimization techniques in Section 4.6. Our results again show that the observed WCL of ZCLLC, ZIV+ROC, and ROC are within the analytical WCL when no optimization technique is deployed (NORMAL configurations). Our computed analytical WCLs as shown in Table 4.4 show that ZIV+ROC and ROC have strictly larger analytical WCL compared to ZCLLC. For NORMAL configuration, Figure 4.7 shows that the observed WCLs of ZIV+ROC and ROC are also larger than or equal to the observed WCLs of ZCLLC for 2, 4, and 8 core configurations across all benchmarks. Meanwhile, the observed WCLs of ZCLLC-NORMAL are larger compared to ZCLLC-OPT. We also observe that, even when GP, ZIV+ROC, or ROC are equipped with all optimizations presented in Section 4.6, the observed WCLs of these configurations can be higher compared to the analytical WCL of ZCLLC-OPT. As Figure 4.7c shows, under 8-core configuration, the observed WCLs of all configurations that are not ZCLLC-OPT exceed the analytical WCL of ZCLLC-OPT for all benchmarks except the water-sp (water spatial for solving molecular dynamics N-body problem).

4.7.2 Performance

We evaluate the performance impact of ZCLLC and the optimizations in Section 4.6 on SPLASH-3 workloads.

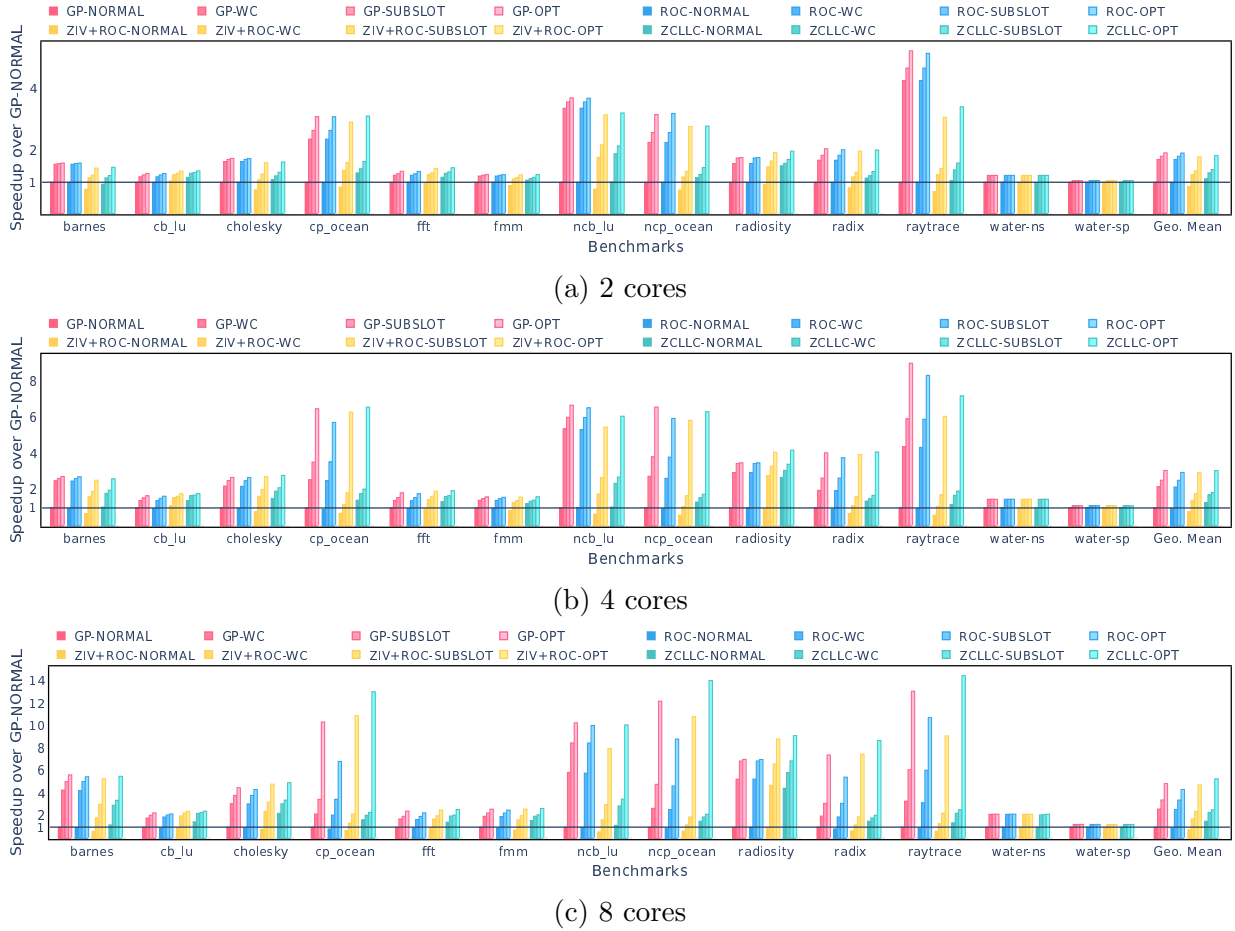


Figure 4.8: The speedups are shown for the following configurations: GP, ROC, ZIV+ROC, and ZCLLC with 2, 4, and 8 cores when running SPLASH-3 benchmarks. Optimizations to these configurations are applied incrementally to all configurations in the order of NORMAL (no optimization), WC, SUBSLOT, and OPT. All speedups are against the GP-NORMAL configuration.

NORMAL configurations. We observe that ZIV+ROC, and ZCLLC show a performance benefit compared to ROC even for its NORMAL configuration, where no optimization technique is used. In this paragraph, we refer to a configuration without specifying their -NORMAL for brevity. Figure 4.8 shows the speedup of ZCLLC, ZIV+ROC, and ROC over the general purpose coherence mechanism where no special mechanism guarantees timing predictability. For 2, 4, and 8 cores, ZCLLC shows a geometric mean of

speedup of $1.16\times$, $1.39\times$, and $1.66\times$ compared to GP. While the ROC shows a geometric mean of speedup of $1.00\times$, $0.97\times$, and $0.91\times$ compared to GP. Note that a speedup less than 1 effectively means a slow down. The performance benefit shown in ZCLLC is a result of the removal of back-invalidations. Moreover, we observe that employing ZCLLC does not cause performance degradation compared to GP. Thus, ZCLLC preserves the performance benefits of ZIV+ROC while not suffering from the enlarged WCL.

Impact of optimizations. When incrementally applying the optimizations in Section 4.6, we observe an improvement in performance across 2, 4, and 8-core configurations. The increasing geometric means presented in Figure 4.8a, Figure 4.8b, and Figure 4.8c highlight such observation. The speedup geometric mean is normalized to GP-NORMAL and averaged across the SPLASH-3 benchmarks. For example, for 8-core ZCLLC configuration, the geometric means of speedups are $1.66\times$, $2.40\times$, $2.66\times$, $5.71\times$ for ZCLLC-NORMAL, ZCLLC-WC, ZCLLC-SUBSLOT, and ZCLLC-OPT, respectively. The increasing speedups when applying optimizations incrementally are also presented in GP, ZIV+ROC, and ROC mechanisms. Notably, for 8-core configurations, with all the optimizations, our results show that ZCLLC-OPT shows speedups of up to $1.89\times$, $3.36\times$, and $6.24\times$ when compared to the state-of-the-art technique of sharing LLC partitions (ROC-NORMAL) for 2, 4, and 8 cores. These speedups are calculated by dividing the unrounded geometric mean of speedups of ZCLLC-OPT by ROC-NORMAL. The performance benefit of ZCLLC-OPT over GP-NORMAL is due to the combination of optimizations, which improves the utilization of the LLC, the command bus, and the data bus. We emphasize that while the optimizations in NORMAL can be integrated into other predictability mechanisms and general-purpose configurations, and present performance improvements, its benefits to the lowered WCL are unique to ZCLLC.

We also evaluate the average memory access latency for different configurations as shown in Figure 4.9 for 2, 4, and 8 cores. Figure 4.9 shows that when employing all performance optimizations (-OPT), the average latencies for ZCLLC are 2.3, 2.2, and 2.3 cycles respectively for 2, 4, and 8 cores. In GP configuration with all optimizations, the average latencies are 2.0, 2.1, and 2.6 cycles for 2, 4, and 8 cores respectively. When no optimization is deployed (-NORMAL), ZCLLC shows average latencies of 5.9, 10.9, and 20.9 cycles for 2, 4, and 8 cores respectively; and the average latencies are 6.6, 13.9, and 30.2 cycles for GP. The reduction in average memory access latency in OPT is a result of relaxing the constraint that a TDM slot is large enough to complete one memory request, and allowing memory requests from different cores to overlap. Notably, all configurations have higher average memory access latencies than an ideal value of 1 cycle due to cache misses.

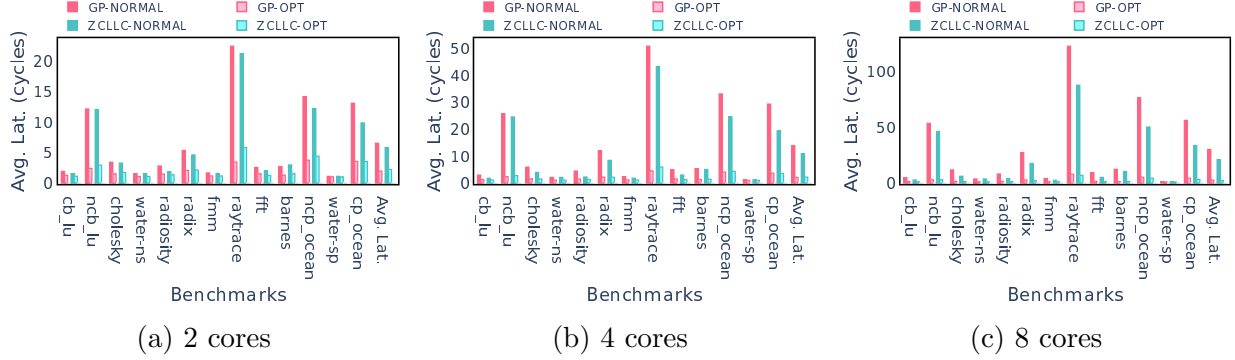


Figure 4.9: The average memory access latencies are shown for both GP and ZCLLC. Configurations equipped with all optimizations (GP-OPT and ZCLLC-OPT) consistently achieve lower memory access latencies compared to configurations without optimizations (GP-NORMAL and ZCLLC-NORMAL).

4.7.3 WriteBack-induced Main Memory Accesses

In an inclusive LLC, as the execution proceeds, the LLC will be filled with dirty cache lines written back by the private caches. As a result, Q in VINV increases, and it is more likely for the LLC to perform an UpdateMemory (Line 18 in Algorithm 1) to maintain VINV when processing WriteBack requests from the cores. We use a smaller LLC size of 256kB to ensure that UpdateMemory is stressed when processing WriteBack. Figure 4.10 validates this observation by showing the fraction of WriteBack that is converted into UpdateMemory. Since ROC and ZIV+ROC do not incur WriteBack when processing UpdateMemory, their fractions will always be 0. For 2-core, 4-core, and 8-core setups that exhibit UpdateMemory, the fractions of WriteBack converted to UpdateMemory can be as high as 97.8%, 98.2%, and 99.3%, respectively. Figure 4.11 shows the number of main memory accesses normalized to ZCLLC. For 2-core, 4-core, and 8-core setups, the numbers of main memory accesses performed by ZCLLC are $0.98\times$, $0.97\times$, and $1.01\times$ compared to the number of main memory accesses performed by ZIV+ROC on average. Hence, although ZCLLC shows a high UpdateMemory fraction, the total number of main memory accesses does not increase compared to ZIV+ROC.

Relationship to performance. At a high level, the total number of UpdateMemory affects performance directly since more UpdateMemory leads to more main memory transfer. However, a high fraction of WriteBack transformed into UpdateMemory does not indicate an increase in the total number of UpdateMemory. Instead, the fraction indicates the percentage of main memory transfers that are *moved* from Read and Write requests

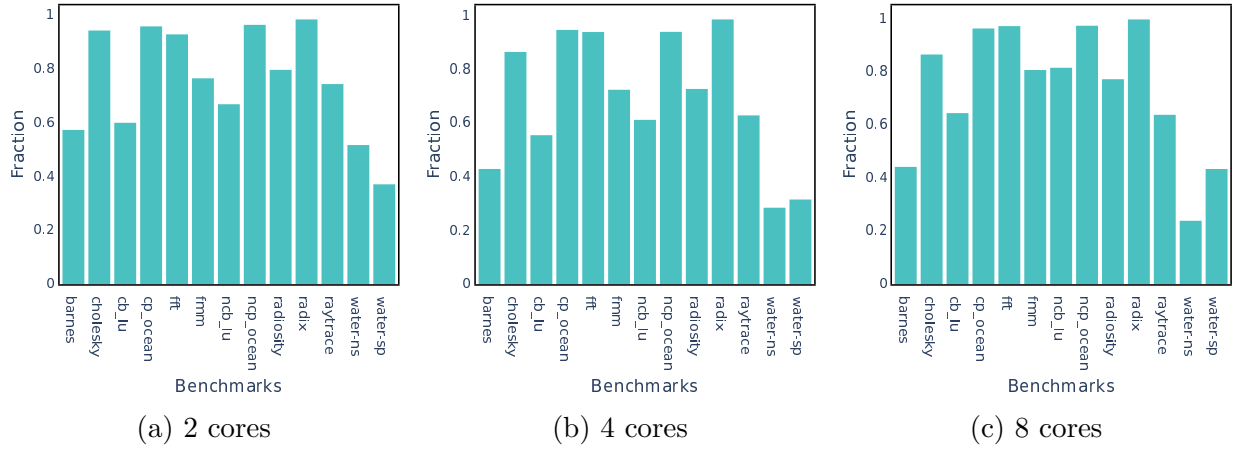


Figure 4.10: The fraction of WriteBack that causes UpdateMemory indicates how many replacements from L2 actually causes access to the main memory in ZCLLC-OPT; we set the LLC size to 256kB to ensure the UpdateMemory is stressed when processing WriteBack.

to WriteBack requests, which is not directly related to the performance. Specifically, the UpdateMemory experienced by ZCLLC for WriteBack would occur in the Read and Write requests in ZIV+ROC or ROC where a UpdateMemory or BI must be employed to create a vacant entry.

Relationship to write-policy. In our opinion, ZCLLC is a write-back cache. Although the UpdateMemory to WriteBack ratio can be high, we would like to point out that the UpdateMemory to WriteBack ratio identifies the ratio of WriteBacks that end up having to update the main memory. This does not suggest whether ZCLLC is either a write-back cache or a write-through cache. Consider the following scenario where we focus on writes to one cache set A in ZCLLC. Suppose that all entries in set A have had prior Writes to them; thus, every entry has valid data that has been modified. Any subsequent Writes to A that miss will require evicting a cache line before completing the request. Since this cache line is dirty with modified data, the main memory would need to be updated with this evicted cache line. In this case, the ratio will be 1.0, but this is not an indication of the LLC being write-through. The update to memory only happens in this scenario because set A is full. Therefore, we feel that ZCLLC is a write-back cache. ZCLLC could behave like a write-through cache when the number of CREs is insufficient for guaranteeing VINV, and it would perform UpdateMemory to create a CRE under this scenario.

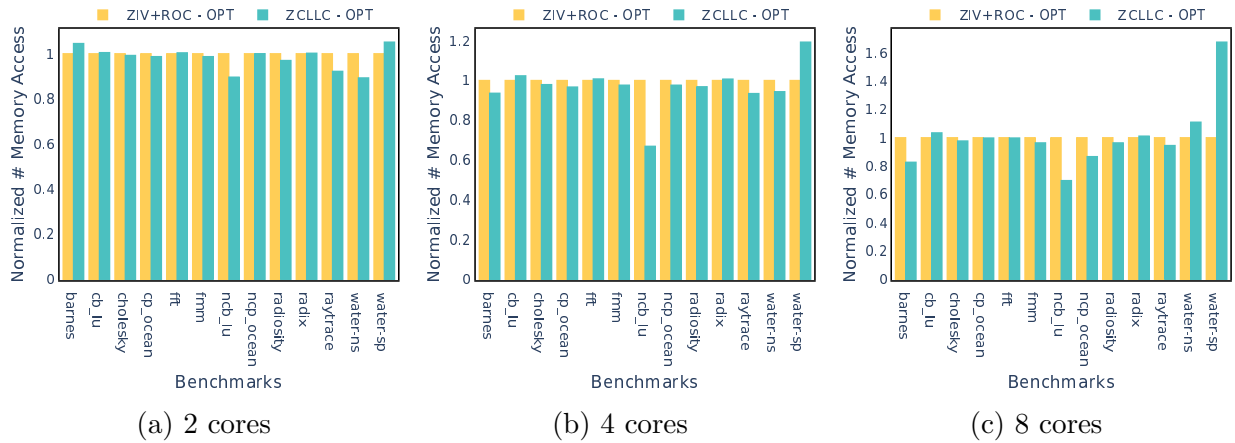


Figure 4.11: The number of main memory accesses are normalized to ZIV+ROC-OPT; we set LLC size to 256kB to ensure the UpdateMemory is stressed when processing WriteBack.

4.8 Related Works

We broadly classify the rich body of research on estimating timing interference with LLCs into two main approaches: (1) methods that use static cache or measurement-based analyses [41, 55, 56, 93, 165], and (2) software and hardware techniques that minimize timing interference when using LLC such as LLC partitioning [46].

Static cache analysis of multi-level caches [41, 55, 56, 93, 165] estimates WCET of tasks by performing must-may and persistence analyses. However, these analyses assume there is no inter-core interference in the shared cache, which is in practice achieved by cache partitioning. ZCLLC does not impose any such constraints on the LLC. Further, with ZCLLC, we provide a per-request WCL that we envision could be used as an input to static cache analyses approaches. Building this into a WCL analysis framework is reserved for future work.

Cache partitioning [46] is often used to partition the LLC on predictable real-time multicore platforms. The main benefit of LLC partitioning is that it removes inter-core or inter-task timing interference by isolating a portion of the LLC to a core or task. As a result, LLC partitioning simplifies timing analyses [8] by allowing it to leverage existing single-core analysis approaches without having to incorporate the effect of accesses made by other cores. Cache partitioning can be implemented in software through page coloring mechanisms managed by the operating system [20, 46, 83, 162] or in hardware through hardware logic managed by the LLC controller [66, 83, 131]. A recent approach called cache

bleaching [118] reduces address fragmentation and run-time recoloring overhead when page coloring is used to perform LLC partitioning. Cache bleaching uses programmable logic to change the addresses of requests from LLC before the requests arrive to the main memory, such that addresses of the same color map to a contiguous region in the main memory, allowing efficient recoloring in the OS without costly data migration. ZCLLC is orthogonal to cache bleaching when page coloring is used to partition the LLC.

Realizing the importance of data sharing between real-time tasks, authors in [29,83] proposed techniques to enable data sharing between real-time tasks using producer/consumer buffers and wait-free buffers implemented in the LLC and main-memory. However, in order to eliminate the interference caused by back-invalidations, these works impose constraints on their caching behavior such as locking the buffers in the LLC (in order to prevent their eviction and hence, back invalidation) and disallow caching of buffers altogether in the cache hierarchy. On the other hand, ZCLLC takes a novel approach in eliminating back-invalidations without imposing any data caching constraints.

Prior works have proposed novel cache designs that combine properties of write-through and write-back cache designs to improve average-case performance [90,92] and energy efficiency [30,37,112]. However, none of these approaches focused on eliminating back-invalidations in the cache hierarchy. As a result, these prior approaches incur high WCL despite offering better performance and energy efficiency. On the other hand, ZCLLC combines properties of write-through and write-back cache designs to manage write-backs from the LLC to main-memory in order to create CREs and eliminate back-invalidations. As a result, the WCL of a memory request under ZCLLC is lower than proposed approaches. ZCLLC can be extended with the insights from [92] to further improve average-case performance and with the insights from [30,37,112] to improve energy efficiency.

4.9 Summary

In this chapter, we present ZCLLC-WC (ZCLLC), a shared inclusive LLC that does not incur additional cost to the WCL of memory requests when added to the memory hierarchy without an LLC. In addition, we systematically analyze the arbitration used in ZCLLC-WC, and observe that the fixed slot width, and the coupled arbitration of command bus and data bus lead to inefficient use of the buses and the LLC. We overcome these inefficiencies by using dynamically changing slot width and splitting the arbitration between the command and data bus. We further introduce a write buffer to eliminate the blocking at the LLC when splitting the bus arbitration. ZCLLC-WC and ZCLLC-OPT can be shared and does not suffer from the limitations of LLC partitions. With all the optimizations,

our results show that ZCLLC-OPT shows speedups of up to $6.24\times$ when compared to the state-of-the-art technique of sharing LLC partitions.

Chapter 5

Consistency-aware and Predictable Memory Processing for Safety-critical Out-of-order Multicores

Apart from the integration of last-level cache introduced in previous chapters, allowing cores themselves to issue multiple outstanding memory requests is another way to exploit memory level parallelism (MLP) in the memory hierarchy to achieve high performance. Traditional wisdom dictates that only one outstanding memory request per core is allowed to maintain predictability and deliver worst-case access latency (WCL) bounds on memory requests. The incarnation of such approaches are (1) predictable in-order pipelines such as [53, 95], (2) predictable cache coherence mechanisms such as [63, 75, 76, 78], and (3) predictable main memory controllers [149], and interconnects [44, 145]. These approaches, however, divert from the practices in general-purpose systems to achieve high performance, where non-blocking caches and out-of-order pipelines are used to exploit MLP.

Recent interests in adopting out-of-order pipelines [48, 49] in safety-critical embedded systems show the possibilities of achieving timing predictabilities for adopting general-purpose high-performance techniques. However, the lack of support for multicore in these works limits their applicability to safety-critical embedded systems, where their adoption in the industry is moving towards multicore systems [5].

In this chapter, we overcome these drawbacks by building multicores that adopt out-of-order pipelines for cores, a predictable cache coherence mechanism between cores' caches, non-blocking caches for higher performance, and timing predictability with a low worst-case latency (WCL) on memory accesses. By timing predictability, we mean **delivering**

a guaranteed upper bound on WCL of memory accesses. To achieve this, we enable cores to issue multiple outstanding memory requests, allowing them to take full advantage of various micro-architectural performance techniques. However, two key challenges must be addressed. (1) We need to ensure that multithreaded programs honour a memory consistency model, which was not necessary when restricting each core to only one outstanding request per core as done in state-of-the-art related works. (2) We must design a technique that guarantees a WCL, and offers a lower WCL when compared to employing existing techniques to ensure memory consistency. We address these challenges in predictable processing of multiple outstanding requests (PPP), which enables out-of-order cores, non-blocking caches, and a hardware cache coherence mechanism while preserving the TSO consistency model and timing predictability. Our main contributions in this chapter are as follows.

1. We identify that introducing memory-level parallelism naively leads to memory consistency model violations.
2. We examine approaches used in general-purpose systems to ensure the memory consistency model and show that those approaches result in a large WCL empirically.
3. We propose micro-architecture extensions, collectively called PPP, that ensure both TSO and *timing predictability*. The resulting multicores have out-of-order pipelines, non-blocking caches, and predictable hardware cache coherence. PPP integrates hardwares that monitor for potential memory consistency model violations. On detecting a violation, PPP delays the request until the request does not causes memory consistency model violations.
4. We provide a WCL analysis for PPP.
5. We implement PPP in gem5 [21] simulator, and compare PPP against two mechanisms that enforce TSO: serialization of memory requests, and retry memory requests.

Benefits of PPP. PPP offers three benefits over state-of-the-art safety-critical multicores.

1. **Performance.** PPP directly addresses the performance limitations in state-of-the-art safety-critical multicores caused by only allowing single outstanding memory request per core in the memory hierarchy. PPP enables multiple outstanding memory requests per core for safety-critical multicores. Our results show that PPP achieves a speedup of $2.07\times$, $2.79\times$, and $3.38\times$ compared to serialization for 2, 4, and 8 cores

configurations, while maintaining the analytical WCL bound. Moreover, PPP provides a comparable performance to general-purpose implementations with only a 5% performance loss.

2. **Functional Correctness.** PPP enforces sequential consistency and total-store order memory models. This functional correctness requirement is non-negotiable even in general-purpose multicores; failure in doing so, such as integrating prior works naively [103], is a hardware bug. This is because failing to honour a consistency model results in unexpected outputs for multithreaded programs.
3. **Low Worst-Case Latency.** PPP delivers a lower worst-case latency than its counterparts while offering performance comparable to general-purpose approaches. We observed the worst-case latency of retry and general-purpose approaches are up to $6.0\times$ and $5.4\times$ of the observed worst-case latency of PPP.

5.1 Background

We introduce necessary background information to understand the challenges of enforcing memory consistency models in multicore systems in this section.

5.1.1 Predictable hardware cache coherence

Predictable hardware cache coherence is a state-of-the-art data-sharing mechanism for safety-critical embedded system multicores, allowing cores to privately cache data with a consistent view of the data across all cores and offering high performance [58, 62, 63, 103]. Predictable hardware cache coherence ensures each memory request has a bounded worst-case latency (WCL) [58, 63, 103]. Predictable hardware cache coherence operates at the granularity of cache lines, typically 32-byte or 64-byte. We refer to the cache line whose first byte is at address A as: *cache line A* or simply A . An example of predictable hardware cache coherence is the MSI-based protocol, PMSI* [105]. PMSI* assigns each cache line a state: **M**odified, **S**hared, or **I**nvalid. A cache line in **M** state is privately cached by only one core and this core can read from or write to the cache line. A cache line in **S** state can be privately cached by one or more cores. Finally, a cache line is in **I** state if the core does not privately cache the cache line.

c_1	c_2	c_1	c_2
(i1) $[C] = 1$	(i4) $[B] = 1$	(i1) $[C] = 1$	(i4) $[B] = 1$
(i2) $r_1 = [A]$	(i5) $[A] = 1$	(i2) $r_1 = [C]$	(i5) $[A] = 1$
(i3) $r_2 = [B]$	(i6) $r_3 = [C]$	(i3) $r_2 = [B]$	(i6) $r_3 = [C]$
Forbidden in SC and TSO: (a) $r_1 = 1, r_2 = 0, r_3 = 0$		Forbidden in SC, allowed in TSO: (b) $r_1 = 1, r_2 = 0, r_3 = 0$	

Figure 5.1: (a) The example shows results disallowed in SC and TSO. (b) Forwarding value from (i1) to (i2) causes execution result disallowed in SC.

5.1.2 Memory consistency model

A *memory consistency model* describes the correctness of shared memory accesses for multithreaded programs. Specifically, the memory consistency model identifies the values loads return when executing multithreaded programs, and the eventual state of the memory. Consequently, a memory consistency model defines the expected order of memory accesses seen by all cores. To help our exposition, we introduce several definitions that formalize SC and TSO memory consistency models. This formalization will help readers understand the challenge in enforcing memory consistency models in multicore systems, and allows us to provide an understanding of correctness behind PPP.

Definitions and Terminology. We focus on memory operations of a multithreaded program when running on a set of N cores $\mathbb{C} = \{c_1, c_2, \dots, c_N\}$. Hence, we denote the set of load and store operations of a multithreaded program as $\mathbb{L}$ and $\mathbb{S}$, respectively. To represent initial values in memory, we augment $\mathbb{S}$ with initial stores s_0^a that populate the initial state of the memory at the address a . Throughout this chapter, we assume the initial state in memory to be 0 for all addresses, unless specified otherwise. For a memory operation $o \in \mathbb{O}$, $\mathbb{S}|_o$ identifies the set of stores to the address targeted by the o . We start by describing the program order of a program executed by a specific core c in Definition 6. The main intuition behind this definition is that $\xrightarrow{p(c)}$ is a total order on the operations issued by core c , allowing us to determine which operation comes before another in the program.

Definition 6 (Program order per core). *The program order of a program to be executed on core c , denoted as $\xrightarrow{p(c)}$, is defined as:*

1. $\xrightarrow{p(c)}$ is a total order, and

2. $\forall o_1, o_2 \in \mathbb{O}, (o_1.\text{core} = o_2.\text{core} = c \wedge o_1 \neq o_2) \implies o_1 \xrightarrow{p(c)} o_2 \vee o_2 \xrightarrow{p(c)} o_1$. $o_1.\text{core}$ is the core issuing o_1 .

We use $\xrightarrow{p(o)}$ for a memory operation $o \in \mathbb{O}$ to mean the program order of the core that issued the memory operation. We construct the *program order* by combining $\xrightarrow{p(c)}$ from individual cores, as defined in Definition 7. The program order allows us to discuss memory operations from any cores.

Definition 7 (Program order). *Two operations $o_1, o_2 \in \mathbb{O}$ are in program order, i.e. $o_1 \xrightarrow{p} o_2$ iff $\exists c \in \mathbb{C}, o_1 \xrightarrow{p(c)} o_2$.*

Next, we introduce the notion of a memory order in Definition 8 and the closest store in Definition 9. The memory order identifies the order of memory operations that are visible to all cores. Notice that program order is a per-core total order, and memory order is a total order over all operations from all cores; thus, some interleaving of all the memory operations.

Definition 8 (Memory order). *Given memory operations $\mathbb{O}$, a memory order, $\xrightarrow{m}$, is a total order defined on $\mathbb{O}$.*

Consider the example in Figure 5.1(a), which shows a multithreaded program. Recall that all initial values in memory are 0. Core c_1 executes one store and two load operations: (i1) writes the value 1 to memory address C , (i2) reads from memory address A into register r_1 , and (i3) reads from B into r_2 . Core c_2 executes two stores and a load: (i4) and (i5) write the value 1 to memory addresses B and A , respectively, and (i6) reads C into a register r_3 . Here, $(i1) \xrightarrow{p} (i2) \xrightarrow{p} (i3)$ for c_1 and $(i4) \xrightarrow{p} (i5) \xrightarrow{p} (i6)$ for c_2 illustrate the program orders. For the memory order, an execution where all operations of c_1 execute in program order followed by all operations of c_2 , also in program order, results in $(i1) \xrightarrow{m} (i2) \xrightarrow{m} (i3) \xrightarrow{m} (i4) \xrightarrow{m} (i5) \xrightarrow{m} (i6)$. This memory order returns the value 0 for r_1, r_2 , and 1 for r_3 . Another execution may execute c_2 's operations in program order before c_1 's resulting in the value 1 for r_1, r_2 and 0 for r_3 . The memory order for the second possible execution is $(i4) \xrightarrow{m} (i5) \xrightarrow{m} (i6) \xrightarrow{m} (i1) \xrightarrow{m} (i2) \xrightarrow{m} (i3)$. Both sets of values for the registers are possible depending on how memory operations are viewed by all cores. We compactly represent the operations in program order, and memory order using $\langle \cdot \rangle_p$ and $\langle \cdot \rangle_m$. For example, $\langle i1, i2, i3 \rangle_p$ is the program order of operations for core c_1 , and $\langle i4, i5, i6, i1, i2, i3 \rangle_m$ is the memory order for the second possible execution.

A memory consistency model has a value function that identifies the values returned when a core performs a load. Therefore, the value function maps each load to the store

that it obtains its value from. Defining the value function requires defining the *closest store*, which identifies the store that wrote the value that a load reads from. We formalize the *closest store* to a load operation in Definition 9. This definition takes a total order t and the load l as inputs, and identifies the store from which the load operation obtains its value. If such a store does not exist, then the load takes on the initial value. When we use a total order as a parameter or an argument, we refer to it without the underlined arrow. Thus, we use t for the total order $\xrightarrow{t}$, p for $\xrightarrow{p}$, $p(o)$ for $\xrightarrow{p(o)}$, and m for $\xrightarrow{m}$.

Definition 9 (Closest store). *Given a total order $\xrightarrow{t}$ on memory operations, the **closest store** to a load operation $l \in \mathbb{L}$ is defined as:*

$$C(t, l) = \begin{cases} s, s \in \mathbb{S}|_l \wedge \forall s' \in \mathbb{S}|_l, s' \neq s \implies \neg(s \xrightarrow{t} s' \xrightarrow{t} l), \\ s_0^{l.addr}, \text{ otherwise} \end{cases} \quad (5.1)$$

Here, the closest store operation s to a load l is the store where $s \xrightarrow{t} l$, and there is no other store s' to the same address that occurs between s and l in the total order.

Using the closest store definition, we can define the value functions. However, different memory consistency models have different value functions based on the constraints imposed by the particular memory consistency model. In Definition 10, we show that for SC, the load returns the value from the most recent store in the memory order.

Definition 10 (Value function for SC). *Given a program order $\xrightarrow{p}$, and a memory order $\mathbb{O}$ of the program, $\xrightarrow{m}$, the value function for the sequential consistency memory consistency model is defined as:*

$$V_{SC}(p, m, l) = C(m, l)$$

For TSO, as described in Definition 11, a load gets its value from the most recent store to the same address. However, this store may be in the store buffer of the core that issued the store; thus, its value would not have been propagated to the memory hierarchy. This means that other cores would observe the older value, but loads to the same address from the core that issued the store would return the value written to by the store. Such store is captured by $C(p(l), l)$, which is the closest store to l in the program order of the core issuing l . Note that TSO enables the store buffer optimization, which is the reason for the difference between SC and TSO.

Definition 11 (Value function for TSO). *Given a program order $\xrightarrow{p}$, and a memory order $\mathbb{O}$ of the program, $\xrightarrow{m}$, its value function is defined as follows:*

$$V_{TSO}(p, m, l) = \begin{cases} C(p(l), l), & l \xrightarrow{m} C(p(l), l) \\ C(m, l), & \text{otherwise} \end{cases}.$$

A multithreaded program can have multiple memory orders. This is because cores see values in the memory via loads; however, the order in which these loads occur can vary based on their executions by their respective cores. Even so, two or more memory orders can have loads return the same values for all load operations. We determine these memory orders to be equivalent. Definition 12 formalizes this insight as an equivalence relation. This requirement complies with SC and TSO, and complies with the architecture assumed in this work.

Definition 12 (Memory order equivalence). *Given a memory consistency model $k \in \{SC, TSO\}$, a program order $\xrightarrow{p}$, and the value function, V_k , two memory orders $\xrightarrow{m_1}$ and $\xrightarrow{m_2}$ are **equivalent** iff they return the same values for all loads, and the orders of stores to the same address are the same. Formally, $m_1 \equiv_k m_2$ iff*

1. $\forall l \in \mathbb{L}, V_k(p, m_1, l) = V_k(p, m_2, l)$, and
2. $\forall s_1, s_2 \in \mathbb{S}, s_1 \xrightarrow{m_1} s_2 \iff s_1 \xrightarrow{m_2} s_2$.

We now define the meaning of a memory order honouring a particular memory consistency model. Definition 13 shows that a memory order is compliant with SC when operations in program order also appear in the same program order in the memory order. We also say the memory order is in SC to denote that it is compliant with SC. There are four possible program orderings of operations: Load followed by a Load (load-to-load), Load followed by a Store (load-to-store), Store followed by a Load (store-to-load), and Store followed by a Store (store-to-store). A memory order is in SC when values returned by the memory order are the same as an equivalent memory order where all four program orderings are preserved. Consequently, the interleaving of memory operations in the memory order preserves the orderings of each individual core's program, and the values returned are the same.

Definition 13 (Compliance with SC). *Given memory operations $\mathbb{O}$ and a program order $\xrightarrow{p}$, a memory order $\xrightarrow{m}$ of $\mathbb{O}$ complies with the SC iff*

$$\exists m', m \equiv_{SC} m', \forall o_1, o_2 \in \mathbb{O}, o_1 \xrightarrow{p} o_2 \implies o_1 \xrightarrow{m'} o_2.$$

Recall that TSO allows store buffer optimization, where a load operation obtains the value written by a store (to the same address) before the store's value is written to the memory hierarchy. Hence, unlike SC, TSO does not require the store-to-load order to be preserved in the memory order. Therefore, a memory order is TSO when operations in the memory order are in program order, except for when there is a store operation that is ordered after a load operation to the same address.

Definition 14 (Compliance with TSO). *Given memory operations $\mathbb{O} = \mathbb{S} \cup \mathbb{L}$ and a program order $\xrightarrow{p}$, a memory order $\xrightarrow{m}$ of $\mathbb{O}$ complies with TSO iff*

$$\begin{aligned} & \exists m', m \equiv_{TSO} m', \\ & \forall o_1, o_2 \in \mathbb{O}, o_1 \xrightarrow{p} o_2 \implies o_1 \xrightarrow{m'} o_2 \vee (o_1 \in \mathbb{S} \wedge o_2 \in \mathbb{L}). \end{aligned}$$

Notice in both Definitions 13 and 14, loads are always in program order in the memory order. Thus, load-to-load ordering is always preserved in either of these memory consistency models. Our work allows violating the load-to-load ordering for higher performance, except when a potential memory consistency model violation (non-consistent values) may occur.

Example of TSO. Consider the example shown in Figure 5.1(b), where the *difference* with the program in Figure 5.1(a) is that (i2) in c_1 loads from C instead of A . Recall that TSO relaxes the ordering of store-to-load operations, and allows loads to read values from the store buffer to dependent loads from the same core. Let us assume that an execution of this example results in a memory order of $\langle i6, i2, i3, i1, i4, i5 \rangle_m$ and the value for r_1 is 1 and 0 for r_2 and r_3 . This memory order is TSO because all orderings except for store-to-load are preserved with the following execution. Suppose that core c_1 executes all its operations in program order, causing the store of C , (i1), to be inserted in the store buffer. Since (i2) loads C , it can read the value 1 from the store buffer. (i3) can then proceed before the store to C has propagated to the memory hierarchy. Similarly, core c_2 's (i4) and (i5) are entered in the store buffer for core c_2 . Then, (i6) can proceed before A, B get propagated to the memory hierarchy. Hence, (i3) and (i6) can proceed before the stores in the store buffers write their values to the memory, causing the loads to return the value 0. This memory order is TSO, but not SC. It is not SC because (i2) is older (comes before) in the memory order than (i1); thus, (i2) should have read the value 0, but it did not.

Disallowed load reordering in TSO and SC. We wish to allow reordering of loads in hardware, but only as long as the reorderings comply with the memory consistency model. When a pair of loads are reordered in a memory order, and such reordering violates the memory consistency model, we refer to this as a disallowed load reordering. To understand how a violation occurs when such reordering happens, consider again the example in Figure 5.1(a). We will explain how the value of 1 for r_1 and 0 for r_2 will result in non-compliance with TSO. The central observation is that (i2) and (i3), and (i4) and (i5) must execute in program order because store-to-store ordering is maintained in TSO. If (i2) returns the value 1, then (i5) must have written 1 to A . Also, since (i4) and (i5) must execute in program order, (i4) will write 1 to B . Thus, (i3) must return 1, but it does not.

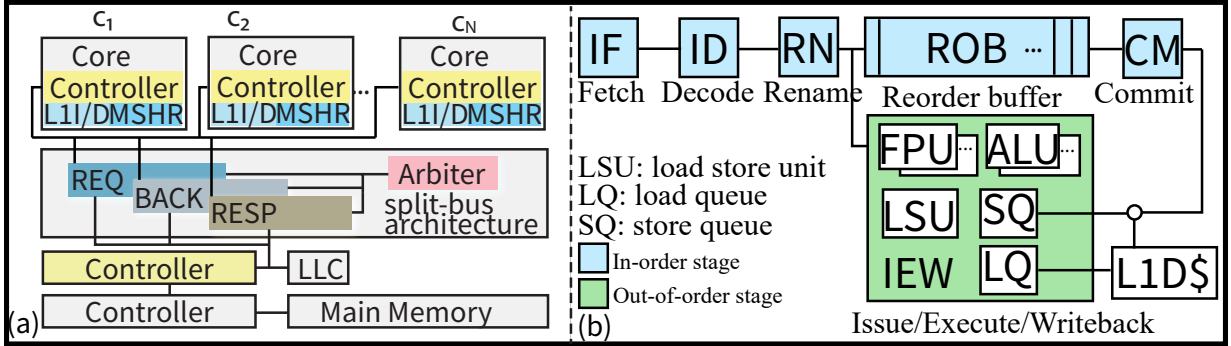


Figure 5.2: System model for PPP assumes that N out-of-order cores and the LLC are interconnected with a split-bus.

The contradiction is evident since (i3) executing after (i2) reads B , which should return the value 1. Note that this violation occurs irrespective of the value taken by r_3 . One memory order leading to such a result is $\langle i3, i6, i4, i5, i1, i2 \rangle_m$. Notice how (i2) and (i3) are reordered, leading to (i3) reading 0 from B . Hence, $r_1 = 1$ and $r_2 = 0$ are not allowed in TSO. This execution result triggers disallowed load reordering of (i2) and (i3) for TSO. A similar argument can be made for SC.

5.1.3 Key takeaways

All state-of-the-art research on predictable hardware cache coherence for safety-critical embedded systems allow only one memory request per core. This disallows an architecture that exploits memory-level parallelism. This work contends that predictable hardware cache coherence can improve performance by exploiting memory-level parallelism exposed by out-of-order cores. However, the predictable hardware cache coherence must be aware of the memory consistency model; both in its implementation and the ensuing WCL analysis. Further, the selection technique must aim to reduce the WCL. Our work is a first step towards integrating out-of-order pipelines with predictable hardware cache coherence while guaranteeing both timing predictability requirements and performance.

5.2 System Model

Our work considers a multicore system with N cores denoted as $c_1, c_2, \dots, c_N$ as shown in Figure 5.2(a). This multicore system uses the state-of-the-art cache coherence designed for

safety-critical systems called PMSI* [76]. PMSI* uses a split-transaction bus architecture as done with prior state-of-the-art efforts [58, 63, 76, 103] to interconnect the multiple cores.

Core pipeline model. Figure 5.2(b) illustrates an out-of-order core pipeline. The execution of an instruction begins by fetching and decoding instructions in program order during the fetch and decode stages. Decoded instructions are then sent to the rename stage, where register renaming occurs. If hardware resources such as arithmetic logic units, floating point units, or the load-store unit for memory instructions are unavailable, the pipeline stalls in the rename stage. The load-store unit handles memory instructions through its load and store queues. Load and store instructions are buffered in the load queue and store queue in program order, respectively, until they are ready to be sent to the cache. Renamed instructions enter the issue/execute/writeback stage, where out-of-order execution occurs once dependencies are met. An instruction entering issue/execute/writeback stage from rename stage is also added to the reorder buffer, a first-in-first-out queue that helps committing instructions in program order. For arithmetic instructions, operands must be ready, while memory instructions require address calculation and, for stores, the data to be stored. The load-store unit calculates addresses and handles cache requests for loads, while stores prepare the address and data but delay sending to the cache. After issue/execute/writeback, instructions wait in the reorder buffer in the commit stage. The pipeline commits instructions in program order. The core *retires* a store instruction by sending it to the cache *after* the store instruction is committed and all older store instructions in the SQ are retired. This means that stores are processed in their program order.

Memory hierarchy – Caches, LLC, main memory. We assume a memory hierarchy with L1 instruction and data caches, a shared last-level cache (LLC), and a main memory. Each core has a private L1 instruction and data cache. Both L1 caches are managed by an L1 cache controller, and the LLC is managed by an LLC controller. The L1 cache controller also manages the *MSHR* to allow non-blocking accesses from the core. We use L1 cache and private cache interchangeably. The LLC is exclusive to the L1; and the LLC is banked to exploit memory-level parallelism. We assume that access to main memory has a bounded WCL [148].

Split-transaction bus architecture. The cores and LLC communicate via a split-transaction bus, like prior works that deliver high performance [58, 63, 103, 105]. The baseline configuration of the split-transaction bus includes two buses: (1) a request bus (REQ) allowing cores to send requests to the LLC and to snoop other cores' requests, and (2) a response bus (RESP) transferring responses from LLC and cores. The order of requests seen by all cores is governed by the REQ bus. Therefore, the broadcast order of requests

on REQ determines the ordering of requests for the entire system. Note that when a core transfers data over RESP, another core can snoop the transferred data, enabling cache-to-cache transfers. PPP includes a Broadcast Acknowledgement Bus (BACK) for the cores to communicate with the arbiter to enforce the memory consistency model.

Arbiter – Oldest-first global arbitration. The access to REQ, RESP and the LLC follows the oldest-first global arbitration scheme inspired by [103]. In this arbitration scheme, the arbiter uses a work-conserving round-robin to pick the oldest request from each core’s MSHR. The access to the LLC and RESP follows the same ordering of requests as that on the REQ bus: a work-conserving oldest-first global arbitration scheme.

5.3 Key Intuition: Memory Consistency Model Violation and Correction with PPP

Out-of-order pipelines execute loads out of program order for higher performance. This is known as speculatively executing loads. Loads delay the execution of later dependent instructions, causing the pipeline to stall. Instead, if other loads can execute before earlier loads, then this improves the pipeline’s utilization [45, 62, 166]. An important consequence of enabling speculative load execution is that the pipeline can benefit from using non-blocking caches [88]. A non-blocking cache allows cores to proceed executing later operations while earlier operations experience cache misses. However, speculatively executing load operations can violate the memory consistency model resulting in non-compliance with the memory consistency model.

Speculative load execution causing memory consistency model violation. We show that speculatively executing loads can result in a memory consistency model violation. We revisit the example in Figure 5.1(a), assuming the pipeline in Section 5.2 with non-blocking caches. Suppose that B and C are the only cache lines privately cached by c_1 and c_2 , respectively. This means that loads from c_1 to B and from c_2 to C will be cache hits. Assume that (i1) and (i2) are stalled due to unmet dependencies or resource constraints. For instance, (i1) can stall in issue/execute/writeback stage because the address computation for C may have earlier dependent operations, or stall in the store buffer waiting to be drained. Similarly, (i2) can stall due to a cache miss. However, by using a non-blocking cache, if (i3)’s dependencies are met, then (i3) can speculatively execute before (i1) and (i2). Since B is cached by c_1 , (i3) is a cache hit and returns the value 0. We term a load such as (i3) that executes out of program order a *speculative load*. Then, c_1 executes (i1) and (i2) in order, after their dependencies and resource constraints are

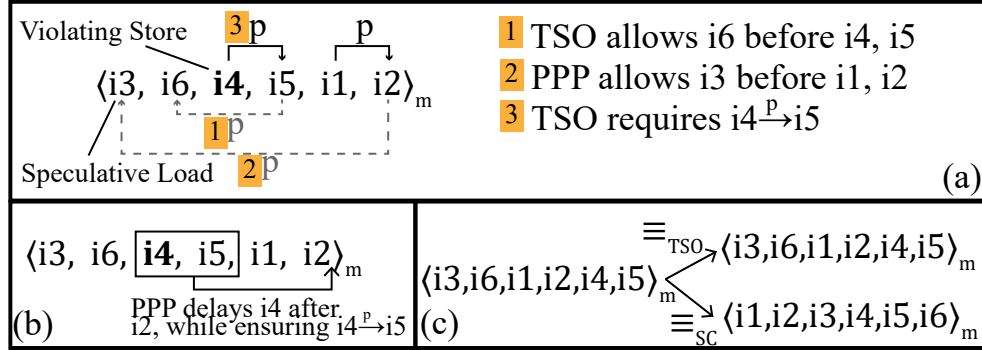


Figure 5.3: (a) Illustration of DLR. (b) PPP postpones violating store. (c) final memory order is equivalent to SC and TSO. Memory operations are from Figure 5.1(a).

resolved. For c_2 , let us assume that $(i4)$ and $(i5)$ are stalled due to dependencies in computing the addresses A and B , respectively. Since TSO allows store-to-load to be executed out of program order, $(i6)$ executes before $(i4)$ and $(i5)$. As C is cached by c_2 , $(i6)$ is a cache hit and returns the value 0. Then, c_2 retires $(i4)$ and $(i5)$ in order. The resulting execution produces $\langle i3, i6, i4, i5, i1, i2 \rangle_m$. Note that since $(i2)$ comes after $(i5)$ in memory order, $r_1 = 1$; and $r_2 = 0$ due to the cache hit on the speculative load $(i3)$. As we explained earlier in section 4.2, this memory order is not SC nor TSO. This shows that speculatively executing loads out-of-order can cause memory consistency model violations when using SC and TSO. Hence, an implementation that promotes speculative execution of loads must prevent such inconsistent results from occurring, and ensure they are not propagated to the architectural state.

Identifying disallowed load reordering conditions. We need to identify conditions under which executing loads out of order violate the memory consistency model. We notice that there are three necessary conditions, as shown below, which we call the *disallowed load reordering conditions* (DLRC). These DLRC guide our design of PPP in the following way. PPP ensures that DLRC never occur; therefore, preventing any memory consistency model violations. In subsequent exposition, we assume memory operations $\odot$ with its program order $\xrightarrow{p}$; we also assume a memory order $\xrightarrow{m}$. Additionally, we will focus on two load operations $l_1, l_2 \in \mathbb{L}$ where they access different addresses: $l_1.addr \neq l_2.addr$.

Definition 15 (DLRC). $\xrightarrow{m}$ violates the memory consistency model due to speculative load execution only if there exists two loads l_1 and l_2 , and these three conditions are met:

1. $\langle l_1, l_2 \rangle_p \wedge \langle l_2, l_1 \rangle_m$,

2. $\exists s_1, s_2 \in \mathbb{S}, \langle l_2, s_1, s_2, l_1 \rangle_m \wedge s_1.\text{core} \neq c \wedge s_2.\text{core} \neq c,$
3. $s_1 \in \mathbb{S}|_{l_2} \wedge s_2 \in \mathbb{S}|_{l_1},$

where core c refers to the core issuing l_1 and l_2 .

We refer to conditions in DLRC as DLRC 1, DLRC 2, and DLRC 3. DLRC 1 states two loads in program order are reordered in the memory order due to speculatively executing loads. DLRC 2 ensures there are at least two stores from other cores between the reordered pair of loads in the memory order. DLRC 3 requires that the store older in memory order writes to the address of the speculative load l_2 , while the second store writes to the address of the other load l_1 . We show the correctness of DLRC in Lemma 5 by contradiction. Our proof assumes that stores are processed in order; this means that the memory order of two stores from the same core complies with their program order, as assumed in our system model.

Lemma 5 (DLRC correctness). *A memory order m violates the memory consistency model due to speculative loads only if there exists two loads that satisfy the DLRC conditions.*

Proof. We prove by contradiction. Suppose for all pairs of loads, DLRC do not hold, then either DLRC 1, DLRC 2, or DLRC 3 fails. If DLRC 1 fails, it means that all loads in m follow their program order; hence, the memory order is compliant with the consistency model. Otherwise, it suffices to show that *when DLRC 2 or DLRC 3 fails and DLRC 1 holds*, we can construct a new memory order equivalent to memory order m that is compliant with the consistency model. We focus on two reordered loads l_1 and l_2 (i.e. $\langle l_2, l_1 \rangle_m$ and $\langle l_1, l_2 \rangle_p$) where there are *no same-core loads between l_1 and l_2 in the memory order m* . Such a pair of loads must *exist* as follows. DLRC 1 implies that there exists a pair of reordered loads l_u and l_v such that $\langle l_u, l_v \rangle_m$ and $\langle l_v, l_u \rangle_p$. If there are no same-core loads between l_u and l_v in m , we can pick $l_1 = l_u$ and $l_2 = l_v$. Otherwise, there exists another load l_w such that $\langle l_u, l_w, l_v \rangle_m$. If $\langle l_v, l_w \rangle_p$, we can pick l_w and l_v , where there are less same-core loads between $\langle l_w, l_v \rangle_m$ than between $\langle l_u, l_v \rangle_m$; this applies equally when $\langle l_w, l_v \rangle_p$ where we can pick l_w and l_v instead. Repeating this process yields a pair of reordered loads l_1 and l_2 with no same-core loads in between. Next, we use case analysis to examine the memory operations between l_1 and l_2 in m , and show that we can rearrange l_1 and l_2 in the memory order to get a new memory order m' that is compliant with the consistency model. The new memory order m' preserves the order in m for all memory operations other than l_1 and l_2 .

Case ► Intermediary operations to addresses other than that of the loads. Suppose all operations between l_2 and l_1 access addresses that are different than l_2 's and

l_1 's, hence failing DLRC 2. These intermediary operations do not affect the value returned by l_2 and l_1 . Hence, l_2 and l_1 obtain their values from stores older than l_2 in the memory order. Therefore, we can simply swap l_1 and l_2 to obtain m' such that $\langle l_1, l_2 \rangle_{m'}$; m and m' are equivalent.

► **Case $\langle l_2, s_1, l_1 \rangle_m$.** Since l_2 reads the value before s_1 writes to it, the write performed by s_1 does not change the value returned by any of the loads. Note that l_1 reads from a different address than s_1 . Thus, the resulting values the loads return are the same as the memory order where $\langle l_1, l_2, s_1 \rangle_{m'}$. The new memory order m' does not change the value function for both SC and TSO because the relative ordering of store operations are unchanged. Hence, $m' \equiv_{TSO} m$ and $m' \equiv_{SC} m$.

► **Case $\langle l_2, s_2, l_1 \rangle_m$.** If the intermediate operation is a store s_2 that writes to l_1 's address, the exact same reasoning applies. The order $m' \equiv_{TSO} m$ and $m' \equiv_{SC} m$ where $\langle s_2, l_1, l_2 \rangle_{m'}$.

► **Case $\langle l_2, s_2, s_1, l_1 \rangle_m$.** Next, another possible memory order is $\langle l_2, s_2, s_1, l_1 \rangle_m$. l_2 loads a value from an older store than s_1 for TSO and SC, or the closest store in program order for TSO; and the same applies to l_1 , which loads a value from s_2 or the closest store in program order for TSO. Hence, l_1 and l_2 still loads the same values in the equivalent memory order $\langle s_2, l_1, l_2, s_1 \rangle_{m'}$. The reason is that such movement does not change the relative order of stores, and hence the values returned by the loads remain.

► **Case $\langle l_2, s_1, s_2, l_1 \rangle_m$.** In this memory order, l_2 loads a value from an older store before s_1 , while l_1 loads a value from s_2 . Note that if this hold for m , it means *all DLRC hold*. Intuitively, l_2 must occur before s_1 for l_2 to return the value before the write of s_1 ; similarly, l_1 must be after s_2 for l_1 to return the value of the write by s_2 . Therefore, there does not exist an equivalent memory order where l_1 and l_2 are in program order while returning the same values. This memory order is not in SC and TSO.

The $\xrightarrow{m'}$, as constructed with the case analysis, preserves all m' order for operations other than l_1 and l_2 , all load values, and still fails DLRC 2 or DLRC 3. In addition, this $\xrightarrow{m'}$ has *one fewer* load reordering than $\xrightarrow{m}$. Note that altering the positions of l_1 and l_2 does not change the values returned by other loads. Hence, after repeating this process, we can find an equivalent memory order with no reordered load by repeatedly reducing the number of reordered loads to zero. This contradicts the initial violation assumption, proving the lemma by contradiction. $\square$

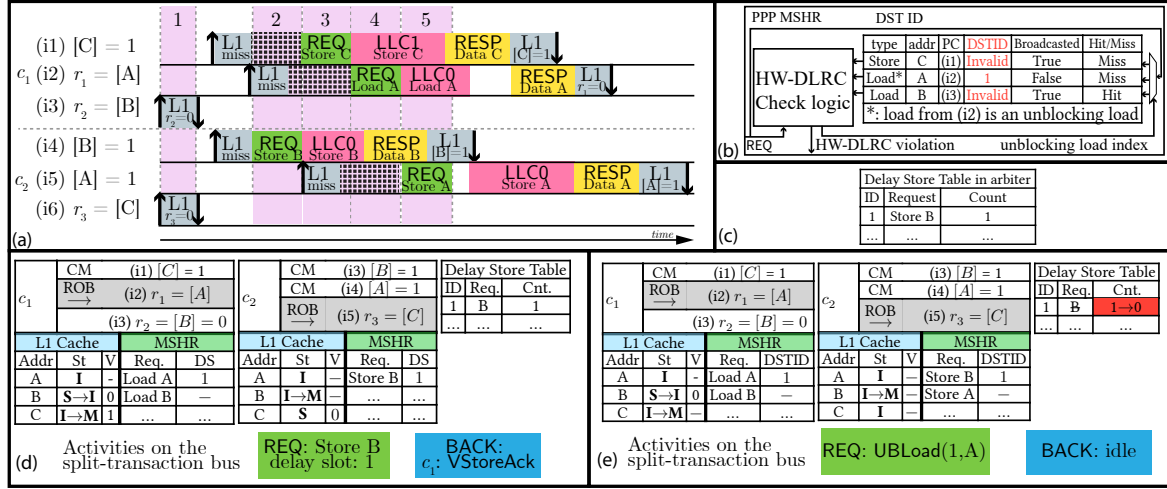
Our approach: Correcting DLRC in PPP. We allow load reordering even if the memory consistency model disallows it, but, when we recognize a potential violation, we

correct it at runtime. Our core insight is that multiple memory orders conform to the memory consistency model, and if we are able to produce one of those equivalent and conforming memory orders, then the result also conforms. Previously, in section 5.1, we showed that $\langle i3, i6, i4, i5, i1, i2 \rangle_m$ is not TSO. This memory order reorders loads ((i3) before (i1) and (i2)), and allows store-to-load reorders with (i6) before (i4) and (i5). While the latter reordering is allowed under TSO, the load reordering is not allowed. Suppose that we are able to detect such a violating memory order before it actually happens, and in response, we produce a corrected memory order $\langle i3, i6, i1, i2, i4, i5 \rangle_m$ instead. A key change in the corrected memory order is that the stores to B and A ((i4) and (i5)) occur after their loads ((i2) and (i3)). Hence, the loads return values that are not updated by stores (i4) and (i5). The corrected memory order is equivalent to $\langle i6, i1, i2, i3, i4, i5 \rangle_m$ where loads are in program order, and only the store-to-load reordering is permitted; thus, complying with TSO.

PPP’s key technique is in providing corrected memory orders when potential memory consistency model violations are identified. We present the main high-level steps that PPP uses to correct potential memory consistency model violations that arise by speculatively executing loads.

- Step 1:** At runtime, PPP speculatively detects if a possible memory consistency model violation may occur due to load reordering. This detection is based on the DLRC conditions. When the detection confirms a possible memory consistency model violation, there exists a store, called the *violating store*, that changes the value being returned by a *speculative load*.
- Step 2:** PPP checks if there are any loads prior to the speculative load in program order that have not yet returned their values. If this check returns true, then PPP delays the violating store until all loads older than and including the speculative load are visible to all cores.

We revisit the memory order $\langle i3, i6, i4, i5, i1, i2 \rangle_m$ that violates TSO and show it in Figure 5.3. We mark the violating store as the store to B (i4) and the speculative load as the load to B (i3). (i3) is the speculative load because it appears before both (i1) and (i2) in the memory order. When (i4) is visible to all cores, each core executes step 1, and detects if a memory consistency model violation may occur. Since (i4) writes to the same address B and (i3) is a speculative load, it is possible to have a memory consistency model violation. Then, in step 2, the core with the speculative load discovers if there are loads in program order before the speculative load that have not returned their values. For this example, (i2) has not returned its value. Thus, the violating store, (i4), is delayed



* Table head in L1 Cache: Addr - address, St - State, V - Value ; Table head in MSHR: Req. - Request, DS - Dependent Stores

Figure 5.4: The working of PPP. (a) A concrete execution of the program in Figure 5.1(a). (b) MSHR is extended with the delay store table index field (DST ID). (c) The delay store table in the arbiter split-transaction bus arbiter. (d) The operation of PPP when an invalidating store broadcasts on REQ. (e) The operation of PPP when the invalidating store is no longer blocked.

until both (i2) and (i3) are visible to all cores. A simple way to visualize the effect of PPP's delaying technique is shown in Figure 5.3, where (i4) and (i5) move in the memory order after (i2). This is the corrected memory order made visible to all cores with PPP at runtime.

5.4 PPP: Hardware Approach to Correct Potential Memory Consistency Model Violations with DLR

We present details of the hardware for PPP to correct potential memory consistency model violations due to speculatively executing loads. A key novelty of our technique is that PPP maintains predictability when correcting potential memory consistency model violations, unlike all prior efforts. PPP introduces two key hardware innovations: (1) a *broadcast acknowledgement bus (BACK)* to detect a violating store, and (2) an orchestrator of operations over REQ, RESP, and BACK to delay the violating store until all speculative loads are completed. We describe the hardware that enables us to detect and delay violating stores,

and then with a concrete example, we show how the hardware is used to correct potential memory consistency model violations. Finally, we introduce a technique that ensures the values returned to loads are correct during the delay of the violating store.

5.4.1 Hardware implementation of DLR correction

The DLR conditions in Section 5.3 determine when load reordering violates the memory consistency model. However, implementing these conditions directly in hardware is impractical, as they require prior knowledge of the memory order, necessitating visibility of all memory operations across cores beforehand. By then, stores would have already updated memory, and loads would have returned inconsistent values, rendering the identification of violations moot. Thus, we propose a runtime approach to detect potential memory consistency violations without the complete memory order.

Detecting DLR violations. We detect *potential* memory consistency model violations due to load reordering with only one *violating store*, outlined in HW-DLR.

Definition 16 (HW-DLR). *Given two load memory operations l_1, l_2 from core c_1 , and a store operation s from another core, the hardware disallowed load reordering violation conditions (HW-DLR) are as follows. (1) $\langle l_1, l_2 \rangle_p$ and l_2 is inserted into memory order, but l_1 is not. (2) $\langle l_2, s \rangle_m$ and (3) $s \in \mathbb{S}|_{l_2}$.*

HW-DLR 1 indicates l_2 executes speculatively before l_1 . HW-DLR 2 states that store s executes after l_2 , and is visible to all cores. HW-DLR 3 means that s writes to the same address as l_2 . These conditions are stronger than DLR conditions because it only uses one store to identify a potential memory consistency model violation. Hence, HW-DLR could identify a possible memory consistency model violation when there is none. Nonetheless, this technique is essential for a practical hardware implementation. The soundness of PPP follows in the following lemma and theorem.

Lemma 6. *PPP ensures that three conditions DLRC will not hold simultaneously in a memory order.*

Proof. Proof by contradiction. Suppose all DLRC hold in a memory order, this means that there is a pair of load l_1 and l_2 and two stores s_1, s_2 such that DLRC 1, DLRC 2, and DLRC 3 hold. However, for DLRC 1 and DLRC 3 to hold, both HW-DLR 1 and HW-DLR 2 must hold for l_1, l_2 , and s_2 . As such, the PPP mechanism engages and prevents s_2 from being broadcasted until l_1 is broadcasted, preventing DLRC 2 and DLRC 3 from holding, contradicting that DLRC hold. $\square$

Theorem 11 (soundness of PPP). *When PPP is deployed, it ensures no memory order violation.*

Proof. Lemma 6 shows that DLRC do not hold with PPP. With contraposition of Lemma 5, PPP ensures no memory order violation. $\square$

5.4.2 Hardware implementation to delay violating stores

PPP requires three hardware components to support delaying violating stores, and returning correct values to loads dependent on such stores. A violating store begins as a cache miss, prompting the cache controller to broadcast a request to fetch the accessed line on REQ. All cores see the REQ broadcast, and all except the issuing core check for any speculative loads reading from the same address. If such a load exists, then the cache controller identifies if there are *other* unbroadcasted loads that are older in program order than the speculative load. When older loads exist, the store is a violating store, and all cores and the arbiter coordinate to delay the violating store. PPP needs hardware support to implement this ability to delay stores and to enforce the memory consistency model.

Delay store table (DST) and REQ bus. PPP implements an N entry delay store table (DST) in the arbiter that records information about stores identified as violating stores. One entry corresponds to delaying one violating store. An entry records an identifier (ID), the address targeted by the violating store, and a Count of the number of cores that identified a violating store to be delayed. There are N entries because each core can delay at most one store at the same time. This is because the core must wait until the violating store completes before proceeding to the next memory operation. Figure 5.4(c) shows an example of the DST. The DST shows a violating store to address B with one core identifying it as a violating store. When the arbiter grants access to a core to broadcast a store on REQ, the arbiter checks the DST for a vacant entry. The ID of the DST entry is broadcast together with the store.

MSHR modifications. Non-blocking caches use MSHR to buffer memory requests that miss in the core’s private cache. The MSHR records the type, address (addr), the broadcast status of the operation, and instruction (instr) as shown in Figure 5.4(b). The type and address indicate the memory operation type (load or store) and the targeted address. The instr field denotes the instructions that bind to the memory operation. The broadcast field is true when the request is placed on REQ. This means all cores have seen this request. PPP extends the MSHR with a field to represent DST ID. The DST ID field is a list of identifiers from the arbiter’s DST of the violating stores in the DST. Figure 5.4(b) shows

an example of an MSHR with three memory operations. Only the load to A has its DST ID marked with a 1, and its broadcast to false. This 1 corresponds to the ID 1 in the DST in Figure 5.4(c) for the store to B signifying that the store to B may cause the speculative load to A to return an inconsistent value for the memory consistency model. We annotate this load as an unblocking load. An unblocking load *un-blocks* a delayed violating store so that the store can proceed¹.

The HW-DLR check logic component implements the detection of a possible memory consistency model violation (HW-DLR conditions) and updating the DST ID entry in the MSHR. Our implementation does the following. When the check logic observes a store on REQ, it initiates the check for HW-DLR. The first step checks the MSHR for loads that have the same address as the store broadcasted on REQ (HW-DLR 2 and 3). This is the load to B for (i3). Since this load is in the MSHR, and it has been broadcasted, HW-DLR 2 is satisfied as the store broadcasted after the load. Then, the check logic searches loads older than the load to B in program order that have not been broadcasted. This identifies speculative loads older than the load of B (HW-DLR 1), which we call the unblocking load. When there are multiple speculative loads, the hardware selects the youngest of these loads and marks it as an unblocking load. The load to A is a speculative load as it has not been broadcasted. The address matching process, done in parallel, is feasible due to the MSHR's small size; similar techniques are also used in silicon-proven processors [110] for efficient store-to-load forwarding within a single clock cycle.

New Coherence Messages. PPP introduces four new messages to coordinate the delaying of violating stores, as shown in Figure 5.5. The use of these messages follows while detailing required hardware structures.

Broadcast Acknowledgement Bus (BACK). PPP extends the interconnect with a BACK bus that private cache controllers use to communicate with the arbiter that a violating store should be delayed. Cores use the VStoreAck message to inform the arbiter that the most recent store is a violating store.

Cache controller modifications. Figure 5.5 lists the new coherence messages used in PPP. When a core observes a violating store, it sends VStoreAck over the BACK bus to the arbiter, indicating that the store broadcast on REQ is a violating store; hence, a DST entry should be allocated.

Recall that the load to A is the unblocking load. When an unblocking load is ready

¹A non-blocking cache, on the other hand, describe the property of the cache where it does not block the core from issuing other memory operations. Hence, an unblocking load and a non-blocking cache are not the same and are at different levels of abstraction.

Message Type	Channel	Explanation
Store (A, s)	REQ	Store request has address A and free delay table entry s
UBLoad (s, A)	REQ	Unblocking load to address A with delay table entry s
VStoreAck	BACK	Acknowledging that the store broadcast on REQ is a violating store
LoadOV	REQ	The Load request that the LLC should return the data value before the violating store

Figure 5.5: PPP extends coherence messages to manage the delaying of violating store and to ensure correct value forwarding.

to be broadcasted on REQ, the cache controller sends an UBLoad message with the address and the DST ID on REQ instead of a regular load request. UBLoad notifies the arbiter to decrement the `count` field in the DST entry of the violating store. When the `count` field reaches zero, the violating store is no longer blocked and the arbiter allows the store to proceed. Note that this means that the arbiter now grants the access of the BANK and RESP to the store.

When a violating store is delayed, other cores can still issue loads to access the cache line written by the violating store. Note that since the violating store is delayed, and not yet visible to all cores (not in the memory order), these loads should appear to complete prior to the violating store. Hence, the loads must return the value of the cache line before the violating store writes to it. Since the violating store is a private cache miss, the value *before* the violating store is in the main memory or is cached in the LLC; the value written by the violating store is in the store buffer of the core that issues the store. When TSO is deployed, younger loads can return the value before the violating store with store-to-load forwarding. PPP ensures that such loads receive the old value by introducing a new request message named LoadOV (*load old value*). When a load is issued on REQ, and the cache line is accessed by a delayed violating store, instead of broadcasting an ordinary load on REQ, the arbiter intercepts the load request and broadcasts a LoadOV request instead.

5.4.3 Concrete execution

We re-examine the high-level example presented in Figure 5.3. Initially, B is cached in c_1 , and C is cached in c_2 . Other cache lines are not privately cached by any cores. Since c_1 and c_2 executes loads speculatively, (i3) and (i6) are sent to the cache out of program order. Both loads hit in the cache. This concludes the execution during 1. Next, c_1 sends

(i1) and (i2) to the cache, while c_2 sends (i4) to the cache. PPP requires that younger stores must be sent after all older stores are visible to all cores, c_2 hence stalls on (i5) until (i4) broadcasts. Commercial processors implementing TSO also sends younger stores after all older stores are visible to all cores [42]. (i1), (i2), and (i4) miss in the cache.

Next, during 2, the arbiter grants access to c_2 and broadcasts (i4) on REQ. When c_1 receives (i4) on REQ, it observes that (i3) is a speculative load hit; in addition, (i3) is not yet broadcasted on REQ. This leads c_1 to send a VStoreAck on BACK, indicating that (i4) is a *violating store*. Recognizing this fact, the arbiter allocates a DST entry with ID 1 for (i4). Additionally, c_1 's MSHR entry corresponding to (i2) records that the DST entry ID of the invalidating store is 1; this marks (i2) as an unblocking load. This indicates that when (i2) broadcasts, a UBLoad is sent instead of an ordinary load. This is shown in Figure 5.4(d). At the end of 2, since (i4) is broadcasted, c_2 can now issue (i5) to the cache, which misses.

During 3, the arbiter grants access to c_1 due to round-robin arbitration; c_1 's (i1) is c_1 's oldest request, and c_1 broadcasts (i1) on REQ. Although C is cached in c_2 , and (i6) is a speculative load, there is no older load in c_2 that is not broadcasted; hence, (i1) is not a violating store and can proceed. If the retry mechanism was used, (i6) must be retried because C is invalidated by (i1).

Next at 4, the round-robin arbiter on REQ should grant access to c_2 . However, since (i4) is a violating store, c_1 's younger memory operations are delayed, until (i2), the unblocking load, broadcasts. Hence, PPP's arbiter grants access to REQ again to c_1 , whose oldest unbroadcast request is now (i2). When (i2) broadcasts on REQ the arbiter recognizes that it is an unblocking load for delay slot 1 because its **delay slots** field is not empty. Hence the arbiter reduces the **count** field for DST entry from 1 to 0. Since now the **count** field for delay slot 1 becomes 0, the arbiter clears the delay slot and allows (i3) to proceed in 5. Without PPP, the arbiter grants access to c_2 and allows (i5) to proceed before (i2), causing r_1 to load 1, exhibiting DLR.

5.5 Worst-case Latency Analysis

Our worst-case latency analysis computes the latency from when a request becomes the oldest among the inflight requests to when this oldest request receives its response from the memory hierarchy. As such, the worst-case latency of multiple requests can be upper-bounded by the sum of the worst-case latency of individual requests [103].

Our analysis divides the worst-case latency of a request into three parts, which naturally follows the life cycle of a request. Figure 5.6 shows the life cycle of a request that begins

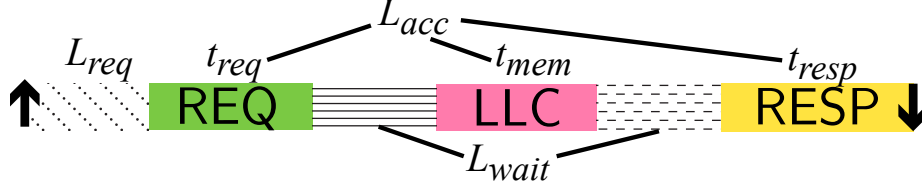


Figure 5.6: Timeline of a request.

with a miss in the private cache and needs to retrieve the data from the LLC. Before this request can be broadcasted over REQ, it suffers L_{req} in the worst case. This is because the round-robin arbitration controls the access to REQ. Next, before the request can access the LLC, and before the request can access RESP, it can be delayed. This is because requests from other cores can be broadcast on REQ before the request under analysis. These older requests access BANK and RESP before the request under analysis. The total latency the request must wait to access the LLC and RESP in the worst-case is bounded by the worst-case resource latency L_{wait} . Accessing the LLC takes t_{mem} in the worst case while accessing RESP takes t_{resp} in the worst case. This means that t_{mem} bounds the worst-case latency of accessing the LLC *and* fetching data from the main memory. The total latency of accessing the resources in the worst case is $L_{acc} = t_{req} + t_{mem} + t_{resp}$.

We identify the critical instance where the request under analysis, r_{ua} , suffers the worst-case latency. We start with the worst-case request latency, which is the worst-case latency it takes for r_{ua} to be broadcasted on the bus.

Lemma 7. *In the worst case, r_{ua} experiences the worst-case request latency when it is ready to be broadcasted on REQ and it is the last among all N cores to broadcast requests. The worst-case request latency is $L_{req} = (N - 1)t_{req}$.*

Proof. In the worst-case, when r_{ua} from c_{ua} is ready to be broadcast on REQ, it is the last in the round-robin order to broadcast; hence it must wait until its turn. Also note that there are $(N - 1)$ cores in the worst-case that must broadcast their requests before c_{ua} can broadcast r_{ua} , where each such broadcast costs t_{req} . $\square$

A request r delays r_{ua} 's access to the LLC or RESP if after r_{ua} broadcasts on REQ, r_{ua} must wait for r 's access to any of the resources to complete (REQ, the LLC, or RESP) before r_{ua} can access the resource.

Lemma 8. *A request r delays r_{ua} 's access to the LLC or RESP if and only if: r is in the MSHR of a core other than c_{ua} , and any of the following conditions is true: (1) r broadcasts*

before r_{ua} on REQ, and, (2) r is a load request that is older than a speculative load to cache line A.

Proof. We first prove the *if* part: If a request completes when r_{ua} broadcasts, since this request will not access the LLC or RESP; hence if a request delays r_{ua} , it must be inflight when r_{ua} broadcasts on REQ, that is, in the MSHR. Because the arbiter maintains the broadcast order of requests, any requests that broadcast before r_{ua} must access the LLC or RESP before r_{ua} , hence a request satisfying (1) would be processed before r_{ua} . Finally, PPP ensures that when r_{ua} invalidates A and there is a speculative load that hits on A, r_{ua} must complete after all prior requests of the speculative load, hence (2). The *only if* part follows from the implementation of the arbiter. Condition (1) follows when r_{ua} is not in the delay slot, the arbiter prioritizes the access to the LLC and RESP for requests that are broadcast before r_{ua} . When r_{ua} is in the delay slot, it must wait until all unblocking load to broadcast on REQ, which incurs condition (2). $\square$

The worst-case resource latency captures the latency that r_{ua} must wait before accessing LLC and RESP.

Lemma 9. *The worst-case resource latency L_{wait} due to waiting for accessing the LLC and RESP is $L_{wait} = (N - 1) \times M \times (t_{mem} + t_{resp}) + (M - 1) \times (N - 1) \times t_{req}$.*

Proof. We can count the number of requests that delay r_{ua} using Lemma 8. In the worst case, the requests that satisfy condition (1) in Lemma 8 is $(N - 1) \times M$. Note that M is the maximal number of outstanding requests. This is because each of the other $N - 1$ cores can broadcast at most M inflight requests when r_{ua} broadcasts its request on REQ. These requests can delay r_{ua} by at most $(t_{mem} + t_{resp})$. In the worst case, the requests that satisfy condition (2) are $(M - 1)(N - 1)$. This is because all requests from all other $N - 1$ cores can have at most $(M - 1)$ requests (excluding the speculative load) that must finish before r_{ua} . Note that in the worst case, all these requests are not broadcast. Hence, these requests can delay r_{ua} by at most t_{req} . $\square$

Theorem 12. *The worst-case latency of a request under analysis r_{ua} is $WCL = L_{req} + L_{wait} + L_{acc}$, where $L_{acc} = t_{req} + t_{resp} + t_{mem}$ is the latency for REQ, RESP, and the LLC to process the request.*

Theorem 12 follows Lemma 7 and Lemma 9.

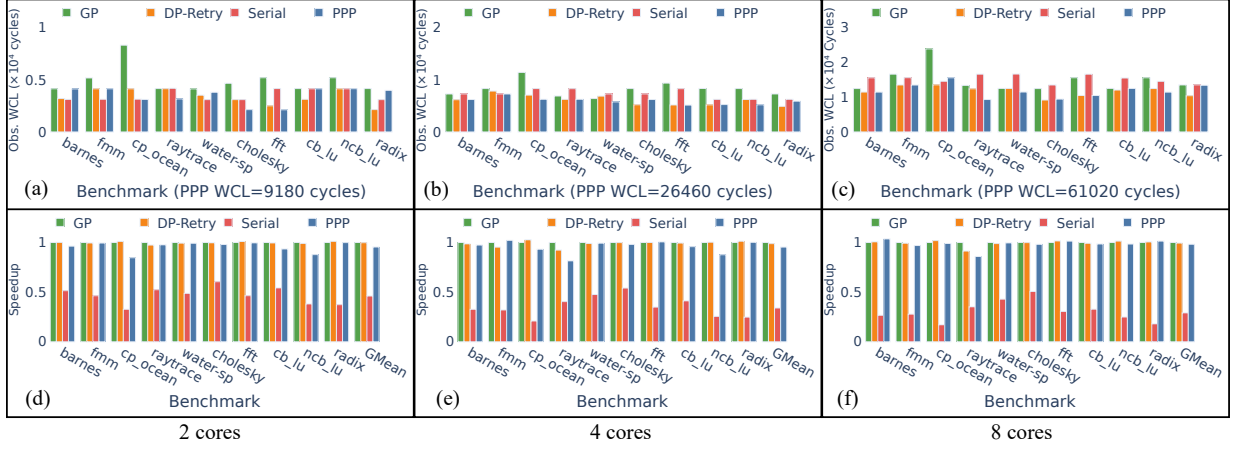


Figure 5.7: Observed worst-case latencies of Serial, DUEPCO, GP, and PPP for 2, 4, and 8 cores are shown in (a), (b), and (c) respectively. Performance of 2, 4, and 8 cores configurations are shown in (d), (e), and (f) respectively. Baseline configuration is GP. The GMean indicates geometric mean.

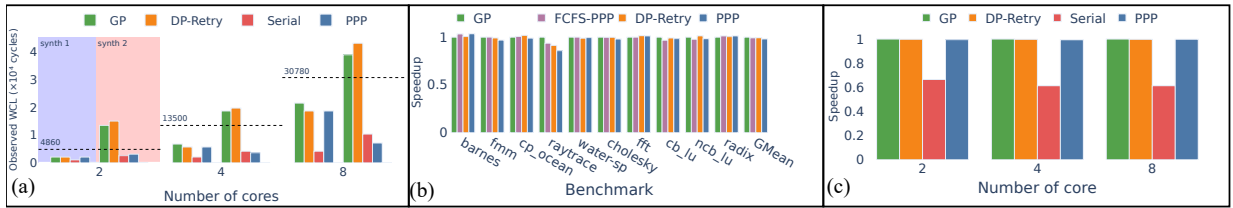


Figure 5.8: (a) Observed WCL of 2 to 8 cores configurations with synthetic benchmarks. The dashed lines tag the WCL of PPP. (b) Speedups of 8-core PPP, FCFS-PPP, and DP-Retry normalized to GP. (c) Running SPLASH-3 benchmarks on 2, 4, and 8 cores configurations with single-threaded execution. Speedups are normalized to GP.

5.6 Empirical Evaluation

We evaluate our proposed approach using the gem5 [21] micro-architectural simulator. We simulate configurations with various number of cores. Each core implements an out-of-order pipeline with an issue width of 8 operating at 2Ghz, and all L1 caches are non-blocking.

We implement five configurations: (1) the **Serial** configuration that serializes all memory requests, (2) **GP** that reflects the **g**eneral-**p**urpose approach of handling memory requests for high performance, where retry mechanism ensures consistency and a first-come-first-serve arbiter arbitrates the memory requests on the split-bus, (3) the prior work on predictable cache coherence that supports *multiple outstanding requests*, DUEPCO, where no special mechanism ensures memory consistency, (4) DP-Retry that deploys the retry mechanism in DUEPCO to ensure memory consistency model, and (5) our proposed approach, PPP. Since each core in **Serial** has only one request, the L1 functions as a blocking cache. Note that this design reflects the handling of memory requests in state-of-the-art out-of-order pipelines [48]. We assume that each core is equipped with 16kB L1 private instruction and data caches. The private caches are backed by a shared 2MB LLC. The private caches and the shared LLC communicate through split buses as presented in the system model. We assume that $t_{req} = 20$ cycles, $t_{resp} = 10$ cycles, and $t_{mem} = 500$ cycles. The choose of t_{req} and t_{resp} accomodate the latency needed by the cores to complete atomic instructions to prevent deadlock in gem5.

5.6.1 Verification of correctness

We exercise our implementation with manually crafted litmus test [101] memory access traces and do not observe TSO or SC violation in our implementation. We further observe that DUEPCO violates SC and TSO, when exercised with the **mp** and **iriw** litmus tests [101]; hence we do not include DUEPCO for further evaluation on performance and predictability.

5.6.2 Worst-case latency

Synthetic benchmarks. Our synthetic benchmarks attempt to stress the different configurations. In all configurations except for **Serial**, we use an MSHR of size 8 ($M = 8$), for efficient creation of the worst-case scenario. **Serial** has an MSHR of size 1 ($M = 1$) to reflect the blocking nature of the cache. The benchmark **synth 1** exercises the WCL of the system by generating memory accesses to the same cache lines. Note that **synth 1** does not exercise the retry mechanism. **synth 2** exercises the memory consistency enforcement

mechanism by timing the requests such that store requests write to cache lines read by speculative loads, causing the retry mechanism to be invoked in both DP-Retry and GP. This triggers both the retry mechanism in DP-Retry and the delay mechanism in PPP.

The results are shown in Figure 5.8(a), where dashed lines and numbers above indicate the WCL of PPP. We observe in **synth 1** that all predictable configurations, including DP-Retry have a WCL that is within the analytical WCL bound. Specifically for PPP, the observed WCL are 2080, 5720, and 18720 cycles for 2, 4, and 8 cores configurations, respectively. In **synth 2**, PPP has a WCL of 3120, 3640, 7163 cycles for 2, 4, and 8 cores, respectively, which are lower than the corresponding analytical WCL bound of PPP of 4860, 13500, and 30780 cycles. We observe that both DP-Retry and GP have a WCL that is higher than the analytical WCL bound of PPP. For example, for 8 core configuration, DP-Retry and GP has a WCL of 43160 cycles and 39000 cycles respectively, higher than PPP’s WCL. In addition, we observe that the total execution time of **synth 2** of DP-Retry and GP is higher than PPP; this is an indication that PPP has a better performance than DP-Retry and GP when the retry mechanism is exercised. Serial shows the lowest WCLs of 4240, 8480, and 16960 cycles compared to PPP and Retry. Despite its low WCL, we will present in following sections Serial’s subpar performance compared to GP and PPP.

Splash-3. We also evaluate the observed WCL with the SPLASH-3 [123] benchmarks. SPLASH-3 consists of a wide variety of complex parallel multi-threaded applications that exercise data sharing and synchronization across cores. We use SPLASH-3 to study the impact of these inter-core interactions on performance and WCL when a system deploys PPP. Figure 5.7(a)-(c) show the results for 2, 4, and 8 cores configurations. Our results show that the observed WCLs of PPP are within the analytical WCL.

5.6.3 Performance

We evaluate the performance of PPP using SPLASH-3 [123]. Figure 5.7(d)-(f) show the results when running SPLASH-3 benchmarks on 2, 4, and 8 cores configurations. In Figure 5.7(d)-(f), we report the *speedup* metric, which is the ratio of the execution time of the baseline configuration, GP, to the execution time of the configuration under evaluation. When compared to the GP mechanism, Serial exhibits slowdown of $2.17\times$, $2.94\times$, and $3.45\times$, for 2, 4, and 8 cores respectively. Hence, enforcing TSO by serializing memory requests incurs a significant performance penalty. On the other hand, PPP has similar performance as GP, where the speedup of PPP over GP is $0.95\times$, $0.95\times$, and $0.98\times$ for 2, 4, and 8 cores respectively. Thus, PPP offers similar performance to a general-purpose scheme to ensure memory consistency while guaranteeing predictability. This is a positive

result of this work. We also notice that DP-Retry has a marginally better performance than PPP. It exhibits a speedup of $0.99\times$ for 2, 4, and 8 cores when compared to GP, such performance comes at the cost of a higher WCL as shown in the previous section. This performance gap is a result of the first-come-first-serve (FCFS) arbiter deployed in DP-Retry. To validate this, we integrate PPP with the FCFS arbiter (denoted FCFS-PPP), and evaluate the performance. Figure 5.8(b) shows that in an 8-core configuration when PPP is integrated with the FCFS arbiter, PPP achieves a speedup of $0.99\times$ over GP. We plan to investigate the impact of integrating PPP with the FCFS arbiter in future work.

Finally, we evaluate PPP’s performance with SPLASH-3 benchmarks when no inter-core interference is present, by running the benchmarks using a single thread for 2, 4, and 8 cores configurations. Figure 5.8(c) shows that PPP has less than 1% performance difference for 2, 4, and 8 cores configurations, respectively, when compared to GP. The close performance between PPP and other mechanisms such as DP-Retry and GP is because in PPP, the delay mechanism is not triggered when there is no inter-core interference. Note that Serial has a slowdown of up to $1.4\times$ compared to GP due to its blocking nature when handling memory requests.

5.6.4 Discussion

This work reveals that PPP guarantees the worst-case latency and maintains SC and TSO while foregoing marginal performance. Unlike DUEPCO, which supports multiple outstanding requests for safety-critical systems but risks producing *logically incorrect results* for memory accesses, PPP ensures *both timing predictability and logical correctness*. Furthermore, PPP delivers performance comparable to GP and DP-Retry in typical execution scenarios, such as those illustrated in Figure 5.7(d)-(f). PPP offers a distinct advantage over others by possessing a worst-case latency bound that is *known* and lower than the observed worst-case latencies of GP and DP-Retry. We envision PPP’s WCL bound could be incorporated into static analyses [48, 99, 147] to provide safe worst-case execution time estimates. Further, PPP’s performance benefit improves the quality of service of applications, and it encourages the deployment of performance demanding applications in safety-critical systems. Also, for systems such as mixed criticality, the slack provided by the performance benefits can be used for other noncritical tasks [137, 139]. As core count increases, PPP’s analytical WCL bounds exceed those of Serial due to increased contention for shared resources. This is an inherent consequence of the parallelism exploited by PPP and does not represent a limitation of the approach; rather, PPP’s WCL remains a safe upper bound.

5.7 Summary

We present PPP, a set of micro-architectural augmentations to enable the use of non-blocking caches, and cache coherence for out-of-order pipelines in safety-critical embedded systems. PPP allows multiple outstanding requests, exposed by both the out-of-order pipeline and the non-blocking caches, for significant performance gains over state-of-the-art techniques that serialize memory accesses by $2.07\times$, $2.79\times$, and $3.38\times$ for 2, 4, and 8 cores. To the best of our knowledge, this is the first work to provide WCL guarantees for memory accesses in safety-critical embedded systems that allow for multiple outstanding requests, while enforcing the memory consistency model.

Chapter 6

Directory-based coherence protocol for predictable cache coherence

Previous chapters, as well as all existing state-of-the-art approaches allowing predictable cache coherent data sharing, focus on the design and implementation of memory hierarchy for safety-critical embedded multicore systems, where cores interconnect through a shared snooping bus. These are apt solutions for multicores with a low core count such as the NXP T2080 that are equipped with cache coherence [113]. However, it is well understood that a shared snooping bus makes scaling to a large number of cores impractical [89]. Nonetheless, addressing this drawback is imperative to meet today’s rising need for additional functionalities, further consolidation of existing functionalities, delivering higher performance, and reducing cost, size, weight, area and power [5, 70]. Consider the Kalray MPPA 3 [13] that has an 80-core architecture, where clusters are connected through NoCs and L2 caches are partitioned across clusters. State-of-the-art approaches cannot be directly applied to such multicore architectures. Therefore, developing a solution that enables predictable cache coherence for larger core counts is paramount for future systems.

In this chapter, we present Ditty: a directory-based cache coherence approach for safety-critical systems that scales to a larger number of cores, guarantees a WCL, and delivers high performance via caching. It is well-known that directory-based cache coherence promote scalability [24]; however, to the best of our knowledge, there is no prior effort in developing a predictable directory-based cache coherence mechanism. With Ditty, we take the first step in doing so. In designing Ditty, we encountered and addressed several important challenges in comparison to efforts that use bus-based interconnects. First, analyses in prior works [63, 75, 76] do not explicitly handle the effect of back-invalidations (a replacement in the LLC requiring evictions of copies that are privately cached) on the WCL when

including an inclusive LLC. Ditty addresses this challenge by explicitly accounting for this in its analysis. Second, naively interconnecting cores using a predictable interconnect such as HopliteRT [145] and employing a general purpose cache coherence protocol [105] may result in a prohibitively large WCL. For this, Ditty methodically identifies scenarios that indeed result in a large WCL, and proposes coherence protocol and micro-architectural additions to lower the WCL.

To summarize, our main contributions in this chapter are as follows.

- We propose a predictable directory-based cache coherence approach for safety-critical systems called Ditty. Ditty employs HopliteRT [145] as its predictable interconnect; however, any predictable interconnect would suffice. Ditty includes carefully considered micro-architectural additions to the L2 cache controller and the directory. In addition, Ditty extends the Modified-Shared-Invalid (MSI) cache coherence protocol to complement the micro-architectural additions.
- We present a timing analysis for Ditty that computes the WCL a memory request can experience. In this analysis, we identify the critical instance, which centers on the eviction of cache lines in the LLC.
- We implement a hardware prototype of Ditty, and deploy it on an FPGA.

6.1 Background and related works

6.1.1 Hardware cache coherence mechanisms

A hardware cache coherence protocol (CCP) dictates the operations on data performed by agents (cores and shared memory) in the memory hierarchy. Cache coherence protocols ensure that all cores observe the same value at a given (logical) time [105]. Figure 6.1a shows the MSI cache coherence protocol [105]. In MSI, each cache line is in one of the following cache states: Modified (M) state, the Shared (S) state or the Invalid (I) state. A cache line in M state means that the contents have been modified by the core and it has read and write access permissions on that cache line. Only one core can be in the M state. S state denotes that a core has read access permission, and the data contents are unchanged. Multiple cores (called sharers) can have a cache line in S to allow read-only access. A cache line in I state means that it is not available in the core’s private cache or its data contents are not up-to-date. M, S, and I states are also called *stable states*.

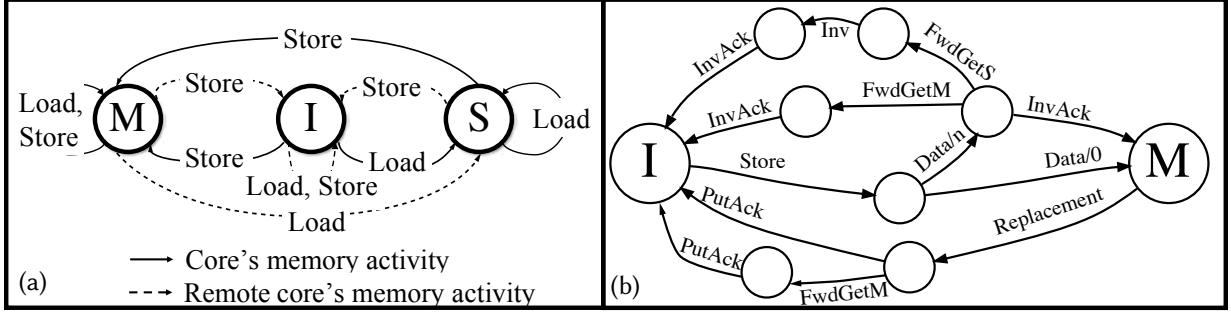


Figure 6.1: (a) MSI cache coherence protocol. (b) The detailed transitions between M and I.

Practical implementations of a CCP use *transient states* to improve performance [105]. These transient states identify that a cache line is transitioning between two stable states. Transient states are essential for overlapping the execution of memory requests. As an example, Figure 6.1b shows the transient states of the transitions between M and I at the core. A core caching a cache line in M needs to invalidate it due to limited set-associativity to serve new requests, causing *replacement* on the cache line. This core transitions from M into a transient state MI, meaning that the cache line is transitioning from M to I. Once the core receives an acknowledgement of the replacement being done (PutAck), it transitions into I. MI allows other cores to access this cache line, and for the core performing the replacement to respond.

Drag away from bus-based approaches. Prior works employ snooping bus-based coherent data sharing, while such an approach enjoys reduced complexity, it suffers from several disadvantages: (1) The bus-based approach has limited scalability [89] as the number of cores increases within a single chip. (2) The bus-based approach must be co-designed carefully with the LLC to manage the adverse impact of back-invalidations, which in the worst-case, can be unbounded [155]. (3) Bus-based approach often deploys global wires, which suffer from low utilization [35]. Moreover, one bus master is granted at a time, hence the overall utilization of the bus is limited. These disadvantages is resolved when migrating towards directory-based data sharing mechanism. Moreover, the use of on-chip interconnect has the benefits during the design cycle since the electrical parameters are known at the earlier stage of the design, enabling aggressive circuit techniques to be used to boost performance or enhance power dissipation [35].

6.1.2 Related works

Directory-based coherence mechanisms are pervasive in general purpose multicore platforms [33, 138]. However, there are no prior efforts in developing a predictable directory-based cache coherence mechanism that can be used in safety-critical systems. Ditty, to our knowledge, is the first to do so.

Most of the recent research activity focused on cache coherence mechanisms for snooping bus-based multicores for safety-critical systems [63, 75, 76, 104]. PMSI [75] utilized a shared command and data bus for communication where an LLC was not present resulting in a WCL that was quadratic in terms of the number of cores. PMSI* [76] proposed coherence protocol changes that lowers the WCL to linear. PISCOT [63] separated the request bus and response bus, where the request bus deploys a time-division multiplexing arbitration and the response bus features a first-come-first-serve arbiter to boost performance. PISCOT’s WCL is linear to the number of cores. Authors in [104] enables predictable coherent data sharing with LLC through the use of a separate invalidation bus and the use of a write-back buffer at the LLC. Authors in [64] deploys COTS coherence protocols on a shared bus deploying predictable arbitration scheme such as TDM, where a direct cache-to-cache interconnect is used to ensure transmission of data within a single TDM slot, achieving a linear WCL. Ditty does not use a shared snooping bus. Instead, it uses a directory-based coherence protocol with a predictable interconnect for transmitting both the command and data, and includes an inclusive LLC.

6.2 System model

Memory hierarchy. Our system model has N cores, a 3-level inclusive cache hierarchy, and a directory connected to a main memory as shown in Figure 6.2. Caches transfer data at 16-byte cache line granularity. L1 and L2 caches are private to the cores and the LLC is shared among all cores. All caches use least-recently-used replacement policy. A directory tracks cache lines privately cached in L2 caches, maintaining coherent memory accesses across all cores by being the ordering point. Existing multicore platforms such as ARM DynamIQ DSU and AMD Opteron platforms [33, 114] use a similar setup. We use MSI to manage coherence between the cores [75]. Table 6.1 shows the messages used for communication by the L2 and the directory. Our analysis focuses on characterizing the WCL when the core issues a request that misses in the L1 and must access L2 to obtain the latency, that is, the worst-case miss latency of L1.

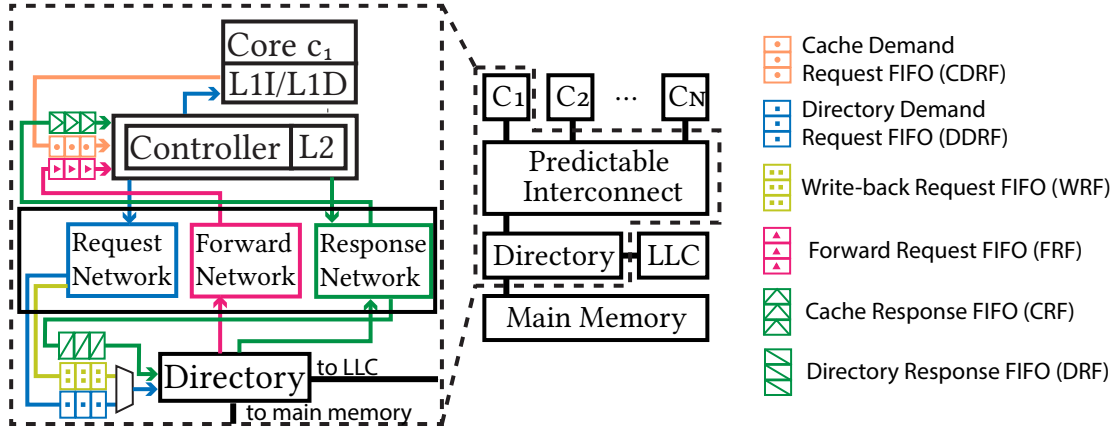


Figure 6.2: Ditty system model.

Interconnect. We require a predictable interconnect to have the following properties: (1) point-to-point communication among connected end-points with three networks, and (2) a worst-case communication latency (WCCL) when sending messages or data between two end-points. General purpose multicores [15, 138] satisfy (1) by using three networks for different types of messages: request, forward, and response. This delivers high performance and ensures correctness [105]. For (2), the WCCL is essential for safety-critical systems and recent efforts include [128, 145]. Note that, for transmission of point-to-point messages in (1), Ditty supports interconnects with point-to-point ordering properties [15, 105] and those without by using acknowledgement messages, making the protocol applicable to point-to-point ordered and unordered interconnect. For our evaluation, we use HopliteRT [145], but Ditty works with any interconnect with the two properties.

FIFOs. Both L2 and the directory use FIFOs to buffer incoming and outgoing messages as shown in Figure 6.2. At the input of the L2 cache controller, there are three FIFOs: cache demand request (CDRF), forward request (FRF), and response (CRF). There are three FIFOs as the input of the directory as well: write-back request (WRF), directory demand request (DDRF), and response (DRF). The CDRF for L2 caches is of size 1. All other FIFOs for L2 caches and the directory are of size N .

► *Core and L2 cache operations.* A core requests data from the memory using a `Load` and writes data to the memory using a `.`. These requests access the L1 cache first. If these requests hit in the L1 cache then the data is returned to the core for a `Load`, and written to the L1 cache for a `.`. If these requests miss in the L1 cache then L2 receives the `Load` and `.` requests on the core's behalf. As in prior works [15, 114, 169], cache coherence operates between the L2 and the LLC. Hence, we focus our discussion on the interaction

Table 6.1: Coherence messages in L2 and LLC

Message	Description	FIFO
Read	Get a shared copy of a cache line	DDRF
Write	Get an exclusive copy of a cache line	DDRF
PutS	Write back a shared copy of a cache line	WRF
PutM	Write back an exclusive copy of a cache line	WRF
FwdGetS	Forward a shared copy of a cache line to another core	FRF
FwdGetM	Forward an exclusive copy of a cache line to another core	FRF
Inv	Invalidate a shared copy of a cache line	FRF
PutAck	Acknowledgement of a write-back request	CRF
InvAck	Acknowledgement of an Inv	CRF, DRF
FwdAck	Acknowledgement of FwdGetS/FwdGetM	CRF, DRF
Data	Data response	CRF, DRF

between the L2s, directory, and the LLC. If `Load` or `hits` in the L2 cache, the requested data is returned to the core via the L1 (following inclusivity). Otherwise, the L2 cache controller issues *demand requests*, a `Read` for a `Load` or a `Write` for a `,` to the directory. `Read` requests the data to be read by the core, and `Write` requests the data that the core plans to modify. Before sending a `Read` or a `Write`, the L2 cache may not have a vacant entry for the requested data and must evict a *victim* cache line. For this, the L2 cache controller issues a `PutS` (clean copy of data) or a `PutM` (dirty copy of data) to relinquish its private copy of the victim cache line. We refer to `PutS` or `PutM` requests as *write-back requests*. Henceforth, we say that a core issues a request or receives a response to mean that the core's L2 cache controller issues a request or receives a response. Table 6.1 shows the usage of FIFOs for each of the coherence messages.

► *Directory operations.* Upon receiving a demand request from the L2 cache controller, the directory orchestrates the necessary communication and sends the requested cache line to the L2 while maintaining coherence of the data between all cores. The directory tracks the owner and a list of sharers of each privately cached cache line. Since L2 is inclusive, only the lines in L2 are tracked. Incoming demand requests and write-back requests are buffered in DDRF and WRF at the directory. The directory prioritizes write-back requests over demand requests. On receiving a `PutS` (write-back of a clean cache line) request for a cache line A from a core c_{ua} , the directory removes c_{ua} from the sharer list of A so that the directory does not have to invalidate the copy in c_{ua} when a later request modifies A . For a `PutM` (write-back of a modified cache line), the directory marks that A is no longer exclusively cached by c_{ua} . The `PutM` carries data modified (dirty) by c_{ua} ; hence, the directory updates the LLC with the most up-to-date data.

6.3 Directory-based cache coherence mechanism

We exhaustively analyzed the MSI protocol and discovered that the design must be done carefully to garner a low WCL. We concentrate our discussion on two key design choices.

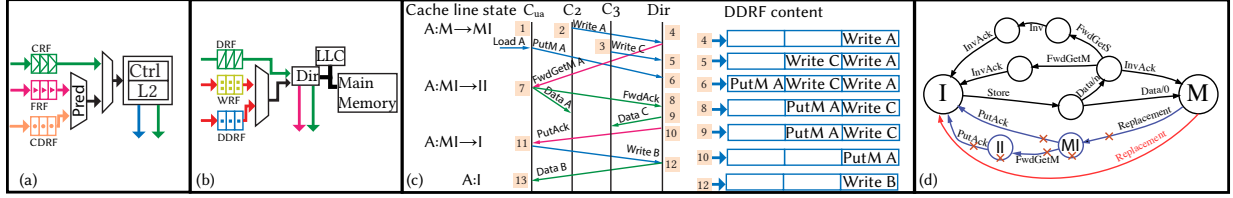


Figure 6.3: (a) Ditty L2 cache contains controller (Ctrl) and storage and has an interface with predictable arbitration. (b) Directory (Dir) interface eliminates PutAck. (c) PutAck introduces long latency. (d) Ditty protocol eliminates PutAck.

6.3.1 L2 cache controller design

In MSI [105], the L2 cache controller arbitrates requests between the forward, demand, and response networks in the following manner: response has highest priority followed by forward and finally demand. This priority order ensures that causal order of messages is preserved. We refer the reader to [105] for more details. Note that this prioritization exists in silicon-proven L2 designs [15]. However, this specific priority order results in a prohibitively large WCL. This occurs when cores request cache lines privately cached by a specific core, c_{ua} . This causes the directory to generate and send forward messages (asks c_{ua} to send data to requesting cores) to c_{ua} 's L2, which get queued in c_{ua} 's FRF. Due to forward messages having higher priority, c_{ua} 's demand request cannot be processed until all forward messages are processed. In the worst-case, a core's demand request is processed by L2 after all cache lines receive two forward requests because of this prioritization. This happens when all cache lines in L2 are in modified state, where the first forward request requests the data be forwarded via a FwdGetS, and the second forward request asks c_{ua} to invalidate its copy via a Inv. Consider an 8kB L2 cache with 512 cache lines, where the processing of each forward request takes 14 cycles. Hence, a core's request is delayed 14336 cycles in the worst-case before being processed by the L2.

Solution 1. The culprit causing the large WCL is the priority order of messages between the three networks. We observe that we can preserve the same properties necessary by the

prioritization [105], but significantly lower the WCL by using a work-conserving round-robin (WCRR) between the demand and forward networks, while the response network remains the highest priority as illustrated in Figure 6.3a. With WCRR, a demand request waits for 14 cycles in the worst-case before it can be processed, and this delay is agnostic of the L2 size.

6.3.2 Coherence protocol: enabling predictable data sharing

MSI implementations require the directory to acknowledge a write-back request with a PutAck message [105]. This is necessary for correctness [86, 105]. Waiting for a PutAck results in a large WCL latency. We explain this with the example in Figure 6.3c with the contents of the DDRF on the right. Suppose core c_{ua} initially caches A exclusively in M state. c_{ua} issues a on B that requires it to evict A as a victim. This causes the PutM on A to be sent to the LLC to revoke its ownership at 1. This transitions c_{ua} 's copy of A to a transient state Ml. Ml indicates that requests for c_{ua} 's copy of A perceive the cache line in M, but after receiving the PutAck, it will be invalid. In parallel, c_2 issues a Write demand request to A at 2, and the directory forwards this to c_{ua} with a FwdGetM. At 3, c_3 issues a Write request to another cache line C , that arrives later than c_2 's request. The directory processes these requests in arrival order due to the FIFOs. Hence, the directory issues FwdGetM to c_{ua} , where c_{ua} sends Data of A as response at 7. Since c_{ua} is still waiting for a PutAck, but it received the FwdGetM request, the cache line state transitions from Ml to transient state Il. This indicates that any further requests to c_{ua} for A will cause c_{ua} to respond with the copy not being valid. Note that by receiving the FwdGetM, c_{ua} knows that another core is writing to A ; hence, it no longer has a valid copy of A . c_{ua} also sends a FwdAck to the directory at 7. The transition from M to I is highlighted in Figure 6.3d. Observe that when the directory receives c_{ua} 's PutM request, the requests by c_2 and c_3 have arrived at the directory before it; thus, delaying c_{ua} 's PutM as shown in the DRF at 6. Hence, the directory must complete c_2 's request for A , and c_3 's request for C before processing the PutM from c_{ua} with a response of PutAck at 10. This allows c_{ua} to issue its next request to B . In the worst-case, all other cores, like c_2 and c_3 , issue requests that arrive at the directory before c_{ua} 's PutM, forcing the WCL to assume $(N - 1)$ requests to be completed before completing c_{ua} 's write-back request. This leads to a large WCL.

Solution 2. Notice that a write-back request is always a result of a demand request. Hence, the demand request must wait for the write-back request to complete before the L2 can perform the demand request. The PutAck signifies the end of the write-back request. Based on this observation, we remove the need to wait for a PutAck on a write-back request

by combining the demand with the write-back request, and sending them together to the directory. This combination also prevents forward requests from intervening the process of the write-back and demand request. This freedom of interference from forward requests lowers the WCL. Ditty augments the coherence protocol and micro-architecture to implement this combining of requests. Figure 6.3d highlights coherence protocol changes for M to I transition, in which a replacement on M directly transitions the line into I without waiting for PutAck.

In the absence of PutAck, a naive implementation that performs all requests in FIFO order may first encounter a demand request to a cache line A cached by a core in modified state. The directory then sends a forward request to the core, asking it to forward the data to both the core and the directory. However, the core may have written back A , and its PutM request is queued after the demand request. The demand request is stalled forever since the directory cannot process the PutM request first. To prevent this situation, Ditty makes micro-architectural modifications as shown in Figure 6.3b. Specifically, Ditty uses two FIFOs, directory demand request FIFO (DDRF) and write-back request FIFO (WRF), to buffer demand and write-back requests, respectively. Additionally, the directory controller prioritizes the processing of write-back requests over demand requests.

6.4 WCL analysis for memory requests

We analyze the WCL a request experiences *after* the request misses in L1 and accesses the L2. Figure 6.4 illustrates the path the request takes in the hardware to experience the WCL. We start with a miss in the L1 resulting in a demand request being sent to the CDRF at the L2 (1). L2 sends this demand request over the request network to the directory (2). This request gets buffered in the DDRF and WRF of the directory (3). When the directory processes this request, it may invalidate copies of the cache line from cores by using the forward and response networks, and fetching the data from the main memory (4). Finally, the directory sends the data to the L2 (5).

We begin by presenting the WCL at the directory (4 in Figure 6.4), WCL_{Dir} using Lemmas 10 and 11.

Lemma 10. *The WCL a memory request experiences at the directory, WCL_{Dir} , is a demand request that is a miss in the directory and causes an eviction of a cache line from the LLC.*

Proof. Note that only write-back and demand requests arrive at the directory. Write-back requests access the directory and LLC’s data array. Demand requests that miss in the

or modified. We omit the case for an invalid victim as its eviction does not incur any interconnect traffic.

Case 1. Victim is shared. A victim line that is shared, in the worst-case, may be privately cached by all N cores. Thus, writing back the victim requires invalidating all N copies due to inclusive cache hierarchy. The directory sends `Inv` messages to N cores and receives acknowledgments incurring a latency of L_{Nmsg} . When an `Inv` arrives at a core's L2, it may be last in its FRF. At worst, this `Inv` message is queued after $(N - 2)$ forward requests in FRF as the `Inv` message occupies one entry in the FRF and each core makes at most one outstanding request. With Solution 1, WCCR ensures the `Inv` is processed after $(N - 1)$ demand requests and $(N - 2)$ forward requests. Notice that a demand request can hit in the L2, whose completion allows a new demand request to appear in the next arbitration cycle. The demand requests, forward requests, and `Inv` itself each take L_{L2} , incurring a latency of $(2N - 2)L_{\text{L2}}$. Finally, it takes L_{1msg} to send back invalidation acknowledgement. When evicting the victim, the directory requests the main memory to fetch the data in parallel, incurring a latency of L_{mem} . This overlapping is possible because the invalidation only occupies the bandwidth of the interconnect and not the bandwidth of the main memory. Hence, the WCL is $\max(L_{\text{Nmsg}} + L_{\text{1msg}} + (2N - 2)L_{\text{L2}}, L_{\text{mem}})$.

Case 2. Victim is modified. When the victim is modified by a core, the directory sends a `FwdGetM` to the core caching the data, causing a latency of L_{1msg} . Similar to the invalidation case, a `FwdGetM` is processed after $(N - 1)$ demand requests and after $(N - 2)$ forward requests in the worst-case, causing a total latency of $(2N - 2)L_{\text{L2}}$. The core then sends the data back to the directory, causing L_{data} . Finally, the directory needs to write-back the data to free an entry in the LLC, causing a latency of L_{mem} . The directory fetches the data in parallel, causing a latency of L_{mem} . Hence, the worst-case latency is $\max(L_{\text{1msg}} + L_{\text{data}} + (2N - 2)L_{\text{L2}}, L_{\text{mem}}) + L_{\text{mem}}$. $\square$

Lemma 12 shows the critical instance of a memory request the core under analysis, c_{ua} , experiences.

Lemma 12. c_{ua} 's request experiences the WCL when (1) the request is processed after a forward request in the L2 cache; (2) the request triggers a replacement of a dirty cache line, incurring a write-back request in the L2; (3) the combined demand request and write-back request undergoes WCCL to reach the directory; (4) the request arrives at the DDRF of the directory and is the last of N demand requests in the DDRF and N write-back requests in the WRF; (5) before the request is processed by the LLC, the directory processes $(2N - 1)$ write-back requests; (6) all N demand requests experience WCL_{Dir} , and, (7) the response undergoes the WCCL to reach the L2.

Proof. We employ Figure 6.4 to guide the proof. (1) A demand request arriving at the L2 in CDRF is assured to be the only request since each core can only have one outstanding memory request. Thus, in the worst-case, this request is processed after one forward request due to the WCRR arbitration (1). (2) When L2 processes this request, the request happens to be a miss. This requires a victim line in L2 to be replaced. However, consider this victim line to be modified. Then, a write-back request (PutM) is necessary to vacate this entry. Thus, the L2 cache controller combines the demand and write-back request, and sends it to the directory. (3) This combined request takes WCCL to arrive at the directory (2). (4) At the directory, write-back requests have higher priority than demand requests; hence, in the worst-case, the PutM of the combined request interferes with its demand request. Further, the directory processes incoming requests from DDRF and WRF one at a time. In the worst-case, the demand request is last in the DDRF. Since each core has at most one outstanding request, all prior $(N - 1)$ demand requests in DDRF have their corresponding write-back requests. Thus, in the worst-case, there are N write-back requests when the combined request arrives at the directory. (5) Before the directory processes the combined request, $(N - 1)$ cores can finish their requests (taking WCL_{Dir} each) that arrived earlier in the DDRF, and they can issue new combined requests. Although the new demand requests are queued later in DDRF, their write-back requests interfere with demand request because the directory prioritizes the write-back requests. At worst, the interference is caused by $2N - 1$ write-back requests. (4), (5) and (6) corresponds to 3 and 4 in Figure 6.4. (7) Finally, the directory sends back the data response incurring WCCL on the response network (5). $\square$

Theorem 13 sums the WCL of a memory request.

Theorem 13. *The WCL of c_{ua} 's memory request is*

$$WCL = 2L_{L2} + L_{data} + WCL_{inter} + WCL_{Dir} + L_{data},$$

where $WCL_{inter} = (2N - 1)L_{Dir} + (N - 1)WCL_{Dir}$ is the interference caused by demand requests and write-back requests at the DDRF and WRF.

6.5 Evaluation

We evaluate Ditty in hardware on a setup with 2, 4, 6, 8, and 10 cores respectively, with cache line size of 16-byte on the Amazon F1 FPGA using Xilinx Vitis 2022.1. We assume a 2-way set-associative L2 cache with a capacity of 512 B, and a 4-way set-associative

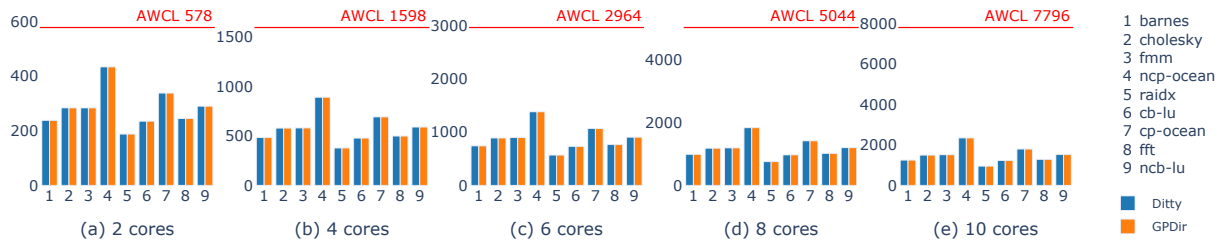


Figure 6.5: Observed WCL in cycles.

inclusive LLC with 32 kB capacity. These capacities ensures 10 cores fit within the FPGA. The cores execute the memory request traces generated by the host machine using the PIN instrumentation tool [98]. Our hardware implementation operates at 100 MHz, and reveals that $L_{L2} = 14$ cycles, $L_{Dir} = 30$ cycles, and $L_{mem} = 200$ cycles. The left part of Table 6.2 shows the interconnect latency components corresponding to each of the configurations.

6.5.1 Synthetic workloads

The synthetic workloads stress our implementation of Ditty by accessing cache lines in the same set in the LLC. We also compare against a general purpose coherent data sharing mechanism (GPDDir). Table 6.2 shows the observed WCL (OWCL) are below the analytical WCL (AWCL) for all configurations.

6.5.2 SPLASH-3 benchmarks

We also evaluate Ditty and GPDDir with SPLASH-3 benchmarks. Figure 6.5 shows the OWCL bound for Ditty and GPDDir. We observe that as the number of cores increases, the WCL exhibited by Ditty is within the AWCL bound. Although our experiments show that GPDDir has similar OWCL, its WCL is not guaranteed as it is for Ditty. Our evaluation indicates that Ditty exhibits less than 0.001% performance difference compared to GPDDir. Although we are able to execute **radiosity** and **raytrace**, due to the large input size, we are unable to finish the execution.

6.5.3 Resource Utilization

Table 6.2 shows the resource utilization of Ditty and GPDDir private caches. Across all configurations, we observe that each Ditty private L2 cache consumes more flip-flops (FFs)

and lookup tables (LUTs) compared to GPDir private L2 cache. This is because Ditty combines demand request and its corresponding write-back request, and hence, the vacant entry is readily available for tracking the demand request. Ditty’s LLC consumes 3114 FFs, 19723 LUTs, 224 BRAMs and 8 URAMs. While the GPDir consumes 2749 FFs, 20902 LUTs, 224 BRAMs and 8 URAMs. Hence, Ditty consumes more FFs compared to a GPDir. This is because Ditty employs DDRF and WRF to store demand requests and write-back requests, while GPDir only has one FIFO for both demand requests and write-back requests.

6.6 Summary

We present Ditty, a predictable cache coherent data sharing mechanism. To achieve predictability, Ditty employs predictable arbitration between demand requests and forward requests at the L2, and proposes to add WRF at the directory. Ditty also eliminates PutAck in the coherence protocol to achieve lower WCL bound. Our empirical evaluation shows that Ditty imposes negligible impact on performance compared to general purpose directory-based coherent data sharing mechanism.

Table 6.2: Interconnect latencies (Intc. Lat.), observed worst-case latency (OWL), analytical worst-case latency (AWL), and resource utilization of Ditty and GPDir. Resource utilization is reported in the number of flip-flops (FFs), lookup tables (LUTs), and block RAMs (BR).

N	Intc. Lat. (cycles)		WCLs (cycles)			Resource utilization					
	L_{1msg}	L_{Nmsg}	Ditty		GPDir	Ditty			GPDir		
			OWL	AWL	OWL	FFs	LUTs	BR	FFs	LUTs	BR
2	15	23	434	578	434	6104	19516	20	7138	20908	20
4	35	59	892	1598	892	12208	39032	40	14276	41816	40
6	63	111	1383	2964	1383	18312	58548	60	21414	62724	60
8	99	179	1845	5044	1845	24416	78064	80	28552	83632	80
10	143	263	2346	7796	2346	30520	97580	100	35690	104540	100

Chapter 7

Conclusion and Future Works

This thesis examines the design and implementation of predictable cache coherence mechanisms for safety-critical systems. The performance and scalability of cache hierarchy of a hardware cache coherence-based predictable shared data communication can be advanced through innovative mechanisms that target interference mitigation, enable dynamic consistency, and adapt to future architectural challenges, while preserving strict timing guarantees.

The key contributions of this thesis include:

1. Identifying that the back-invalidation effect on the WCL is not explicitly handled in prior works [63, 75, 76] when adopting an inclusive LLC. And we present an analysis that explicitly accounts for this effect, showing that the WCL can be bounded for the 1S-TDM arbitration schedule. We also present set sequencer, a hardware structure that lowers the WCL when sharing an LLC partition, cubic in the number of cores.
2. Proposing a novel last-level cache architecture, ZCLLC. ZCLLC completely eliminates back-invalidations from the last-level cache, and thus achieve a WCL that is linear in the number of cores.
3. Proposing a predictable cache coherence approach for safety-critical systems called PPP. PPP allows for multiple outstanding requests per core while also guaranteeing a WCL and ensuring consistency model correctness. PPP achieves this by employing micro-architectural extensions to the shared bus, as well as a novel cache coherence protocol.

4. Proposing a predictable directory-based cache coherence approach for safety-critical systems called Ditty. Ditty employs HopliteRT [145] as its predictable interconnect; however, any predictable interconnect would suffice. Ditty includes carefully considered micro-architectural additions to the L2 cache controller and the directory. In addition, Ditty extends the MSI cache coherence protocol to complement the micro-architectural additions.

The research contributions in this thesis present the following future extensions.

1. **Predictable and High Performance Coherence Protocol for Heterogeneous Safety-critical Architectures:** The slowdown of Moore’s law [39] and the increasing demand for high-performance computing have led to the development of heterogeneous multicore architectures, where domain-specific hardware components such as GPUs [108], FPGAs [157], and accelerators are integrated with general-purpose cores [168]. While the research contributions in this thesis focus on predictable cache coherence for homogeneous multicore architectures, one of the future works is to extend the proposed cache coherence protocols to heterogeneous multicore architectures.
2. **Formal Verification for Predictable Cache Coherence Protocols:** Currently, all the proposed cache coherence protocols are verified using simulation-based techniques and are manually verified. Coherence protocols are complex and are typically verified via theorem provers [134] or model checkers [6]. For safety-critical systems, the cache coherence protocol must not only satisfy the functional correctness as required in general-purpose systems but also guarantee the predictability of the system in the form of a bounded worst-case latency. While tools are available for modeling and verifying real-time systems [19], their application to cache coherence protocols is not studied in the literature. One of the future works is to formally verify the proposed cache coherence protocols using formal verification tools such as Lean [38], or TLA+ [163].

References

- [1] ISO 26262 Standard. <https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en>. Accessed: 2021-10-17.
- [2] Ultra-reliable MPC574XB/c/G mcus for automotive and industrial control and Gateway.
- [3] A. Navarro-Torres et al. Memory hierarchy characterization of spec cpu2006 and spec cpu2017 on the intel xeon skylake-sp. *Plos one*, 14(8), 2019.
- [4] Niket Agarwal, Li-Shiuan Peh, and Niraj K. Jha. In-network snoop ordering (inso): Snoopy coherence on unordered interconnects. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 67–78, 2009.
- [5] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. A comprehensive survey of industry practice in real-time systems. *Real-Time Systems*, nov 2021.
- [6] Homayoon Akhiani, Damien Doligez, Paul Harter, Leslie Lamport, Joshua Scheid, Mark Tuttle, and Yuan Yu. Cache coherence verification with tla. *Edited by G. Goos, J. Hartmanis and J. van Leeuwen*, page 1872, 1999.
- [7] Sebastian Altmeyer, Roeland Douma, Will Lunniss, and Robert I Davis. On the effectiveness of cache partitioning in hard real-time systems. *Real-Time Syst.*, 52(5):598–643, September 2016.
- [8] Altmeyer, Sebastian and Douma, Roeland and Lunniss, Will and Davis, Robert I. OUTSTANDING PAPER: Evaluation of Cache Partitioning for Hard Real-Time Systems. In *2014 26th Euromicro Conference on Real-Time Systems*, volume , pages 15–26, 2014.

- [9] AMD. AMD Technical Information Portal — docs.amd.com. <https://docs.amd.com/v/u/en-US/versal-ai-edge-gen2-psg>. [Accessed 12-01-2025].
- [10] ARM. ARM DynamIQ Shared Unit Technical Reference Manual. <https://developer.arm.com/documentation/100453/0401/functional-description/13-cache/cache-slices-and-portions?lang=en>.
- [11] ARM. AMBA CHI architecture specification. Technical report, 2024.
- [12] ARM. Arm architecture reference manual for a-profile architecture. Technical report, Arm Limited, 2024.
- [13] B. Dupont de Dinechin. Consolidating high-integrity, high-performance, and cybersecurity functions on a manycore. In *ACM/IEEE DAC*, pages 1–4, 2019.
- [14] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):1–25, 2017.
- [15] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrads, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. Openpiton: An open source manycore research framework. *SIGARCH Comput. Archit. News*, 44(2):217–232, mar 2016.
- [16] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrads, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. Openpiton: An open source manycore research framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’16*, page 217–232, New York, NY, USA, 2016. Association for Computing Machinery.
- [17] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrads, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. Openpiton: An open source manycore research framework. *SIGPLAN Not.*, 51(4):217–232, mar 2016.
- [18] Ayoosh Bansal, Jayati Singh, Yifan Hao, Jen-Yang Wen, Renato Mancuso, and Marco Caccamo. Reconciling Predictability and Coherent Caching. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–6, 2020.

- [19] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Up-paal—a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control: Verification and Control*, pages 232–243, Berlin, Heidelberg, 1996. Springer-Verlag.
- [20] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. page 158–170. ACM, 1994.
- [21] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. 39(2):1–7, aug 2011.
- [22] Eric Bost. Hardware support for robust partitioning in freescale qorik multicore socs (p4080 and derivatives). Technical report, Freescale Semiconductor, Inc., 2013.
- [23] Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-core devices for safety-critical systems: A survey. *ACM Computing Surveys (CSUR)*, 53(4):1–38, 2020.
- [24] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, 1990.
- [25] Samarjit Chakraborty and S. Ramesh. Guest editorial special section on automotive embedded systems and software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1701–1703, 2015.
- [26] Mainak Chaudhuri. Zero Directory Eviction Victim: Unbounded Coherence Directory and Core Cache Isolation. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 277–290, 2021.
- [27] Mainak Chaudhuri. Zero Inclusion Victim: Isolating Core Caches from Inclusive Last-level Cache Evictions. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 71–84, 2021.
- [28] Long Chen, Yuchen Li, Chao Huang, Bai Li, Yang Xing, Daxin Tian, Li Li, Zhongxu Hu, Xiaoxiang Na, Zixuan Li, Siyu Teng, Chen Lv, Jinjun Wang, Dongpu Cao, Nanning Zheng, and Fei-Yue Wang. Milestones in autonomous driving and intelligent

- vehicles: Survey of surveys. *IEEE Transactions on Intelligent Vehicles*, 8(2):1046–1056, 2023.
- [29] Chisholm, Micaiah and Kim, Namhoon and Ward, Bryan C. and Otterness, Nathan and Anderson, James H. and Smith, F. Donelson. Reconciling the Tension Between Hardware Isolation and Data Sharing in Mixed-Criticality, Multicore Systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, volume , pages 57–68, 2016.
 - [30] Jongouk Choi, Jianping Zeng, Dongyoon Lee, Changwoo Min, and Changhee Jung. Write-light cache for energy harvesting systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.
 - [31] Laurène Claussmann, Marc Revilloud, Dominique Gruyer, and Sébastien Glaser. A review of motion planning for highway autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, 21(5):1826–1848, 2020.
 - [32] CXL Consortium. Compute express link 2.0 specification. www.computeexpresslink.org.
 - [33] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30(2):16–29, 2010.
 - [34] Henry Cook, Wesley Terpstra, and Yunsup Lee. Diplomatic design patterns: A tilelink case study. In *1st Workshop on Computer Architecture Research with RISC-V*, 2017.
 - [35] William J. Dally and Brian Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the 38th Annual Design Automation Conference, DAC '01*, page 684–689, New York, NY, USA, 2001. Association for Computing Machinery.
 - [36] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4), October 2011.
 - [37] Pepijn de Langen and Ben Juurlink. Limiting the number of dirty cache lines. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 670–675. IEEE, 2009.
 - [38] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 378–388. Springer, 2015.

- [39] Chris Edwards. Moore’s law: what comes next? *Commun. ACM*, 64(2):12–14, January 2021.
- [40] El-Sayed, Nosayba and Mukkara, Anurag and Tsai, Po-An and Kasture, Harshad and Ma, Xiaosong and Sanchez, Daniel. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, volume , pages 104–117, 2018.
- [41] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2):163–189, 1999.
- [42] Jean-Marc Frailong, Pradeep Sindhu, Michel Cekleov, Michael Powell, and Eric Jensen. Method and apparatus for providing total and partial store ordering for a memory in multi-processor system, November 23 1993. US Patent 5,265,233.
- [43] G. Gracioli et al. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2), 2015.
- [44] Tushar Garg, Saud Wasly, Rodolfo Pellizzoni, and Nachiket Kapre. Hoplitebuf: Network calculus-based design of fpga nocs with provably stall-free fifos. *ACM Trans. Reconfigurable Technol. Syst.*, 13(2), February 2020.
- [45] Kouros Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *ICPP’91*, 1991.
- [46] Giovanni Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. 48(2), nov 2015.
- [47] Giovanni Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [48] A. Gruin, T. Carle, C. Rochange, H. Casse, and P. Sainrat. Minotaur: A timing predictable risc-v core featuring speculative execution. *IEEE Transactions on Computers*, 72(01):183–195, 2023.

- [49] Alban Gruin, Thomas Carle, Christine Rochange, and Pascal Sainrat. Enabling timing predictability in the presence of store buffers. In *The 31st International Conference on Real-Time Networks and Systems*, RTNS 2023. ACM, June 2023.
- [50] Daniel Grund, Jan Reineke, and Reinhard Wilhelm. A template for predictability definitions with supporting evidence. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems (2011)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2011.
- [51] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, page 245–254, New York, NY, USA, 2009. Association for Computing Machinery.
- [52] H. Rihani et al. Wcet analysis in shared resources real-time systems with tdma buses. In *RTNS*, page 183–192, 2015.
- [53] Sebastian Hahn and Jan Reineke. Design and analysis of SIC: a provably timing-predictable pipelined processor core. *Real-Time Systems*, 56(2):207–245, November 2019.
- [54] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication Centric Design in Complex Automotive Embedded Systems. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:20, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [55] Damien Hardy and Isabelle Puaut. WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches. In *2008 Real-Time Systems Symposium*. IEEE, November 2008.
- [56] Damien Hardy and Isabelle Puaut. WCET Analysis of Instruction Cache Hierarchies. 57(7):677–694, aug 2011.
- [57] Mohamed Hassan. Discriminative coherence: Balancing performance and latency bounds in data-sharing multi-core real-time systems. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2020.

- [58] Mohamed Hassan, Anirudh M. Kaushik, and Hiren Patel. Predictable cache coherence for multi-core real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–246, 2017.
- [59] Farouk Hebbache, Florian Brandner, Mathieu Jan, and Laurent Pautet. Work-Conserving Dynamic Time-Division Multiplexing for Multi-Criticality Systems. *Real-Time Syst.*, 56(2):124–170, apr 2020.
- [60] Farouk Hebbache, Mathieu Jan, Florian Brandner, and Laurent Pautet. Shedding the Shackles of Time-Division Multiplexing. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 456–468, 2018.
- [61] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.
- [62] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.
- [63] Salah Hessian and Mohamed Hassan. Piscot: A pipelined split-transaction cots-coherent bus for multi-core real-time systems. *ACM Trans. Embed. Comput. Syst.*, 22(1), oct 2022.
- [64] Mohamed Hossam and Mohamed Hassan. Predictably and Efficiently Integrating COTS Cache Coherence in Real-Time Systems. In Martina Maggio, editor, *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, volume 231 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:23, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [65] IEC 2010. IEC 61508(-1/7): Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.
- [66] Intel. Improving real-time performance by utilizing cache allocation technology. *Intel Corporation*, 2015.
- [67] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology - Enhancing Performance via Allocation of the Processor’s Cache. Technical report, Intel Corporation, 2015.

- [68] Intel. Intel® 64 and IA-32 architectures software developer’s manual. Technical report, 2024.
- [69] ISO. ISO 26262-11 Road vehicles—Functional safety, 2018.
- [70] J. Cerrolaza et al. Multi-core devices for safety-critical systems: A survey. *ACM Comput. Surv.*, 53(4), 2020.
- [71] A. Jaleel, M. Mattina, and B. Jacob. Last level cache (llc) performance of data mining workloads on a cmp - a case study of parallel bioinformatics workloads. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 88–98, 2006.
- [72] Norman P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA ’93*, page 191–201, New York, NY, USA, 1993. Association for Computing Machinery.
- [73] Norman P. Jouppi. Cache write policies and performance. *SIGARCH Comput. Archit. News*, 21(2):191–201, May 1993.
- [74] Kalray. White paper coolidge mppa - kalray. <https://www.kalrayinc.com/download/white-paper-coolidge-mppa/>.
- [75] Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems. *IEEE Transactions on Computers*, pages 1–1, 2020.
- [76] Anirudh Mohan Kaushik and Hiren Patel. A Systematic Approach to Achieving Tight Worst-Case Latency and High-Performance Under Predictable Cache Coherence. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 105–117, 2021.
- [77] Anirudh Mohan Kaushik and Hiren Patel. Automatic construction of predictable and high-performance cache coherence protocols for multicore real-time systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(10):3318–3331, 2022.
- [78] Anirudh Mohan Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren Patel. CARP: A Data Communication Mechanism for Multi-core Mixed-Criticality Systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 419–432, 2019.

- [79] R.E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.
- [80] Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 80–89, 2013.
- [81] Hyoseung Kim and Ragunathan (Raj) Rajkumar. Predictable Shared Cache Management for Multi-Core Real-Time Virtualization. *ACM Trans. Embed. Comput. Syst.*, 17(1), dec 2017.
- [82] Hyoseung Kim and Ragunathan (Raj) Rajkumar. Predictable Shared Cache Management for Multi-Core Real-Time Virtualization. *ACM Trans. Embed. Comput. Syst.*, 17(1), dec 2017.
- [83] Namhoon Kim, Bryan C. Ward, Micaiah Chisholm, Cheng-Yang Fu, James H. Anderson, and F. Donelson Smith. Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016.
- [84] D.B. Kirk. SMART (strategic memory allocation for real-time) cache design . In *1989 Real-Time Systems Symposium*, pages 229,230,231,232,233,234,235,236,237, Los Alamitos, CA, USA, December 1989. IEEE Computer Society.
- [85] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.
- [86] Rakesh Komuravelli, Sarita V. Adve, and Ching-Tsun Chou. Revisiting the complexity of hardware cache coherence and some implications. 11(4), dec 2014.
- [87] Adam Kostrzewa, Selma Saidi, Leonardo Ecco, and Rolf Ernst. Flexible tdm-based resource management in on-chip networks. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS ’15, page 151–160, New York, NY, USA, 2015. Association for Computing Machinery.
- [88] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA ’81, page 81–87, Washington, DC, USA, 1981. IEEE Computer Society Press.
- [89] Rakesh Kumar, Victor Zyuban, and Dean M Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *32nd International Symposium on Computer Architecture (ISCA ’05)*, pages 408–419. IEEE, 2005.

- [90] An-Chow Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 139–148, 2000.
- [91] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.
- [92] H.H.S. Lee, G.S. Tyson, and M.K. Farrens. Eager writeback-a technique for improving bandwidth utilization. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pages 11–21, 2000.
- [93] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. WCET Analysis of Multi-Level Set-Associative Data Caches. In Niklas Holsti, editor, *9th International Workshop on Worst-Case Execution Time Analysis (WCET’09)*, volume 10 of *OpenAccess Series in Informatics (OASIs)*, pages 1–12, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6.
- [94] Yonghui Li, Benny Akesson, and Kees Goossens. Architecture and analysis of a dynamically-scheduled real-time memory controller. *Real-Time Systems*, 52:675–729, 2016.
- [95] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES ’08, page 137–146, New York, NY, USA, 2008. Association for Computing Machinery.
- [96] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–766, 2018.
- [97] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, USA, 1st edition, 2000.
- [98] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

- [99] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05:1–05:48, Jun. 2016.
- [100] M. Lv et al. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05:1–05:48, Jun. 2016.
- [101] Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. Litmus tests for comparing memory consistency models: how long do they need to be? In *Proceedings of the 48th Design Automation Conference, DAC '11*, page 504–509, New York, NY, USA, 2011. Association for Computing Machinery.
- [102] Mancuso, Renato and Dudko, Roman and Betti, Emiliano and Cesati, Marco and Caccamo, Marco and Pellizzoni, Rodolfo. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, volume , pages 45–54, 2013.
- [103] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Parallelism-Aware High-Performance Cache Coherence with Tight Latency Bounds. In Martina Maggio, editor, *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, volume 231 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:27, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [104] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Parallelism-Aware High-Performance Cache Coherence with Tight Latency Bounds. In Martina Maggio, editor, *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, volume 231 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:27, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [105] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. A Primer on Memory Consistency and Cache Coherence, Second Edition. *Synthesis Lectures on Computer Architecture*, 15(1):1–294, February 2020.
- [106] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143, 2012.
- [107] NVIDIA. CUDA Toolkit Documentation. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-coherency-hd>.

- [108] NVIDIA. Nvidia orin series system-on-chip technical reference manual. Technical report, 2023.
- [109] NXP Semiconductors. Ultra-reliable MPC574XB/c/G mcus for automotive and industrial control and Gateway, 2022.
- [110] Ishwar Parulkar, Alan Wood, James C Hoe, Babak Falsafi, Sarita V Adve, Josep Torrellas, and Subhasish Mitra. Opensparc: An open platform for hardware reliability experimentation. In *Fourth Workshop on Silicon Errors in Logic-System Effects (SELSE)*, pages 1–6. Citeseer, 2008.
- [111] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, et al. The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc. *IEEE Micro*, 40(2):53–62, 2020.
- [112] Bahareh Pourshirazi, Majed Valad Beigi, Zhichun Zhu, and Gokhan Memik. Writeback-aware llc management for pcm-based main memory systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(2):1–19, 2019.
- [113] Roger Pujol, Hamid Tabani, Jaume Abella, Mohamed Hassan, and Francisco J. Cazorla. Empirical evidence for mpsoes in critical systems: The case of nxp’s t2080 cache coherence. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1162–1165, 2021.
- [114] Falk Rehm, Jörg Seitter, Jan-Peter Larsson, Selma Saidi, Giovanni Stea, Raffaele Zippo, Dirk Ziegenbein, Matteo Andreozzi, and Arne Hamann. The road towards predictable automotive high - performance platforms. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1915–1924, 2021.
- [115] Renesas. RH850/C1M-AX.
- [116] Renesas. R-Car H3 / M3 User’s Manual: Hardware. Technical report, 2020.
- [117] Renesas. RH850/C1M-AX, 2022.
- [118] Shahin Roozkhosh and Renato Mancuso. The Potential of Programmable Logic in the Middle: Cache Bleaching. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 296–309, 2020.

- [119] Alberto Ros, Trevor E. Carlson, Mehdi Alipour, and Stefanos Kaxiras. Non-speculative load-load reordering in tso. *SIGARCH Comput. Archit. News*, 45(2):187–200, June 2017.
- [120] Alberto Ros, Trevor E. Carlson, Mehdi Alipour, and Stefanos Kaxiras. Non-speculative load-load reordering in tso. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 187–200, New York, NY, USA, 2017. Association for Computing Machinery.
- [121] Alberto Ros and Stefanos Kaxiras. Complexity-effective multicore coherence. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 241–251, 2012.
- [122] S. Altmeyer et al. Evaluation of cache partitioning for hard real-time systems. In *ECRTS*, pages 15–26, 2014.
- [123] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111, 2016.
- [124] C. E. Salloum, M. Elshuber, O. Höftberger, H. Isakovic, and A. Wasicek. The across mp soc – a new generation of multi-core processors designed for safety-critical embedded systems. In *2012 15th Euromicro Conference on Digital System Design*, pages 105–113, 2012.
- [125] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, et al. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, pages 449–471, 2015.
- [126] Alberto Scolari, Davide Basilio Bartolini, and Marco Domenico Santambrogio. A Software Cache Partitioning System for Hash-Based Caches. *ACM Trans. Archit. Code Optim.*, 13(4), dec 2016.
- [127] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer’s model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.
- [128] Zheng Shi and Alan Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Proceedings of the Second ACM/IEEE International*

Symposium on Networks-on-Chip, NOCS '08, page 161–170, USA, 2008. IEEE Computer Society.

- [129] Mladen Slijepcevic, Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. Time-analysable non-partitioned shared caches for real-time multicore systems. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.
- [130] Mladen Slijepcevic, Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. Timing verification of fault-tolerant chips for safety-critical applications in harsh environments. *IEEE Micro*, 34(6):8–19, 2014.
- [131] Shekhar Srikantiah, Mahmut Kandemir, and Mary Jane Irwin. Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors. *SIGARCH Comput. Archit. News*, 36(1):135–144, mar 2008.
- [132] Sriram Srinivasan and William L Walker. Shadow tag memory to monitor state of cachelines at different cache level, 2018. US Patent 10,073,776.
- [133] Nivedita Sritharan, Anirudh Kaushik, Mohamed Hassan, and Hiren Patel. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 433–445, 2019.
- [134] Joseph Stoy, Xiaowei Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In *International Symposium of Formal Methods Europe*, pages 43–71. Springer, 2001.
- [135] Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*, pages 300–303, 2008.
- [136] Vignesh Suresh, Bakshree Mishra, Ying Jing, Zeran Zhu, Naiyin Jin, Charles Block, Paolo Mantovani, Davide Giri, Joseph Zuckerman, Luca P. Carloni, and Sarita V. Adve. Mozart: Taming taxes and composing accelerators with shared-memory. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, PACT '24, page 183–200, New York, NY, USA, 2024. Association for Computing Machinery.
- [137] Ali Syed, Gerhard Fohler, and Daniel Gracia Pérez. Online admission of non-preemptive aperiodic mixed-critical tasks in hierarchic schedules. In *2017 IEEE*

23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pages 1–10, 2017.

- [138] Simon M. Tam, Harry Muljono, Min Huang, Sitaraman Iyer, Kalapi Royneogi, Nagmohan Satti, Rizwan Qureshi, Wei Chen, Tom Wang, Hubert Hsieh, Sujal Vora, and Eddie Wang. Skylake-sp: A 14nm 28-core xeon® processor. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 34–36, 2018.
- [139] Hsiang-Kuo Tang, Parmesh Ramanathan, and Katherine Compton. Combining hard periodic and soft aperiodic real-time task scheduling on heterogeneous compute resources. In *2011 International Conference on Parallel Processing*, pages 753–762, 2011.
- [140] V. Suhendra et al. Exploring locking and partitioning for predictable shared caches on multi-cores. In *DAC*, pages 300–303, 2008.
- [141] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016.
- [142] W. Zhang et al. Static timing analysis of shared caches for multicore processors. *JCSE*, 6(4):267–278, 2012.
- [143] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. Making shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 157–167. IEEE, 2013.
- [144] Ward, Bryan C. and Herman, Jonathan L. and Kenna, Christopher J. and Anderson, James H. Outstanding Paper Award: Making Shared Caches More Predictable on Multicore Platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, volume , pages 157–167, 2013.
- [145] Saud Wasly, Rodolfo Pellizzoni, and Nachiket Kapre. Hoplitert: An efficient fpga noc for real-time applications. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 64–71, 2017.
- [146] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The risc-v instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, May 2014.

- [147] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008.
- [148] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 372–383, 2013.
- [149] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 372–383, 2013.
- [150] Zhuahao Wu, Anirudh Kaushik, and Hiren Patel. Zcllc, 2023.
- [151] Zhuanhao Wu, Marat Bekmyrza, Nachiket Kapre, and Hiren Patel. Ditty: Directory-based cache coherence for multicore safety-critical systems. In *2023 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, 2023.
- [152] Zhuanhao Wu, Anirudh Kaushik, and Hiren Patel. Zerocost-llc: Shared llcs at no cost to wcl. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 249–261, 2023.
- [153] Zhuanhao Wu, Anirudh Kaushik, and Hiren Patel. High performance and predictable shared last-level cache for safety-critical systems. *ACM Trans. Embed. Comput. Syst.*, 23(6), September 2024.
- [154] Zhuanhao Wu, Anirudh Mohan Kaushik, Paulos Tegegn, and Hiren Patel. A Hardware Platform for Exploring Predictable Cache Coherence Protocols for Real-time Multicores. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 92–104, 2021.
- [155] Zhuanhao Wu and Hiren Patel. Predictable Sharing of Last-Level Cache Partitions for Multi-Core Safety-Critical Systems. In *Proceedings of the 59th ACM/IEEE Design Automation Conference, DAC '22*, page 1273–1278, New York, NY, USA, 2022. Association for Computing Machinery.
- [156] X. Wang et al. Swap: Effective fine-grain management of shared last-level caches with minimum hardware support. In *HPCA*, pages 121–132, 2017.

- [157] Xilinx. Versal architecture and product data sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds950-versal-overview.pdf.
- [158] Meng Xu, Linh Thi Xuan Phan, Hyon-Young Choi, Yuhan Lin, Haoran Li, Chenyang Lu, and Insup Lee. Holistic Resource Allocation for Multicore Real-Time Systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 345–356, 2019.
- [159] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. Towards developing high performance risc-v processors using agile methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1178–1199, 2022.
- [160] Xu, Meng and Thi, Linh and Phan, Xuan and Choi, Hyon-Young and Lee, Insup. vCAT: Dynamic Cache Management Using CAT Virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, volume , pages 211–222, 2017.
- [161] Yutaka Yamada, Toru Sano, Yasuki Tanabe, Yutaro Ishigaki, Soichiro Hosoda, Fumihiko Hyuga, Akira Moriya, Ryuji Hada, Atsushi Masuda, Masato Uchiyama, Masashi Jobashi, Tomohiro Koizumi, Takanori Tamai, Nobuhiro Sato, Jun Tanabe, Katsuyuki Kimura, Yoshinari Ojima, Ryusuke Murakami, and Takashi Yoshikawa. A 20.5 tops multicore soc with dnn accelerator and image signal processor for automotive applications. *IEEE Journal of Solid-State Circuits*, 55(1):120–132, 2020.
- [162] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392, 2014.
- [163] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Advanced research working conference on correct hardware design and verification methods*, pages 54–66. Springer, 1999.
- [164] Wei Zhang, Mingsong Lv, Wanli Chang, and Lei Ju. Precise and Scalable Shared Cache Contention Analysis for WCET Estimation. In *In proceedings of Design Automation Conference (DAC)*, pages 1–6, 2022.

- [165] Zhenkai Zhang, Zhishan Guo, and Xenofon Koutsoukos. Handling Write Backs in Multi-Level Cache Analysis for WCET Estimation. RTNS '17, page 208–217, New York, NY, USA, 2017. Association for Computing Machinery.
- [166] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, volume 5, 2020.
- [167] Li Zhao, Ravi Iyer, Srihari Makineni, Don Newell, and Liqun Cheng. NCID: A Non-Inclusive Cache, Inclusive Directory Architecture for Flexible and Efficient Cache Hierarchies. CF '10, page 121–130, New York, NY, USA, 2010. Association for Computing Machinery.
- [168] Joseph Zuckerman, Davide Giri, Jihye Kwon, Paolo Mantovani, and Luca P. Carloni. Cohmeleon: Learning-based orchestration of accelerator coherence in heterogeneous socs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 350–365, New York, NY, USA, 2021. Association for Computing Machinery.
- [169] Joseph Zuckerman, Paolo Mantovani, Davide Giri, and Luca P. Carloni. Enabling heterogeneous multicore soc research with risc-v and esp. In *CARRV 2022*, 2022.