Timing Predictable and High-Performance Hardware Cache Coherence Mechanisms for Real-Time Multi-Core Platforms

by

Anirudh Mohan Kaushik

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

Waterloo, Ontario, Canada, 2021

Examining Committee Membership

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

External Examiner:	Jörg Henkel Professor, Karlsruhe Institute of Technology
Supervisor(s):	Hiren Patel Professor, University of Waterloo
Internal Members:	Rodolfo Pellizzoni Associate Professor, University of Waterloo Nachiket Kapre Associate Professor, University of Waterloo
Internal-External Member:	Kenneth Salem Professor, University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this 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.

Abstract

Multi-core platforms are becoming primary compute platforms for real-time systems such as avionics and autonomous vehicles. This adoption is primarily driven by the increasing application demands deployed in real-time systems, and the cost and performance benefits of multi-core platforms. For real-time applications, satisfying safety properties in the form of timing predictability, is the paramount consideration. Providing such guarantees on safety properties requires applying some timing analysis on the application executing on the compute platform. The timing analysis computes an upper bound on the application's execution time on the compute platform, which is referred to as the worst-case execution time (WCET).

However, multi-core platforms pose challenges that complicate the timing analysis. Among these challenges are timing challenges caused due to simultaneous accesses from multiple cores to shared hardware resources such as shared caches, interconnects, and off-chip memories. Supporting timing predictable shared data communication between real-time applications further compounds this challenge as a core's access to shared data is dependent on the simultaneous memory activity from other cores on the shared data. Although hardware cache coherence mechanisms are the primary high-performance data communication mechanisms in current multi-core platforms, there has been very little use of these mechanisms to support timing predictable shared data communication in real-time multi-core platforms. Rather, current state-of-the-art approaches to timing predictable shared data communication sidestep hardware cache coherence. These approaches enforce memory and execution constraints on the shared data to simplify the timing analysis at the expense of application performance.

This thesis makes the case for timing predictable hardware cache coherence mechanisms as viable shared data communication mechanisms for real-time multi-core platforms. A key takeaway from the contributions in this thesis is that timing predictable hardware cache coherence mechanisms offer significant application performance over prior state-of-the-art data communication approaches while guaranteeing timing predictability.

This thesis has three main contributions.

First, this thesis shows how a hardware cache coherence mechanism can be designed to be timing predictable by defining design invariants that guarantee timing predictability. We apply these design invariants and design timing predictable variants of existing conventional cache coherence mechanisms. Evaluation of these timing predictable cache coherence mechanisms show that they provide significant application performance over state-of-the-art approaches while delivering timing predictability.

Second, we observe that the large worst-case memory access latency under timing predictable hardware cache coherence mechanisms questions their applicability as a data communication

mechanism in real-time multi-core platforms. To this end, we present a systematic framework to design better timing predictable cache coherence mechanisms that balance high application performance and low worst-case memory access latency. Our systematic framework concisely captures the design features of timing predictable cache coherence mechanisms that impacts their WCET, and identifies a spectrum of approaches to reduce the worst-case memory access latency. We describe one approach and show that this approach reduces the worst-case memory access latency of timing predictable cache coherence mechanisms to be the same as alternative approaches while trading away minimal performance in the original cache coherence mechanisms.

Third, we design a timing predictable hardware cache coherence mechanism for multi-core platforms used in mixed-critical real-time systems (MCS). Applications in MCS have varying performance and timing predictability requirements. We design a timing predictable cache coherence mechanism that considers these differing requirements and ensures that applications with no timing predictability requirements do not impact applications with strict predictability requirements.

Acknowledgements

This PhD thesis is a culmination of research work spanning over five years (2015-2021). The academic training and character building crucial to complete this thesis started much earlier under the guidance of several mentors who have guided and led me from naive beginnings to this moment. These mentors have taken different forms – teachers, professors, friends, and family. I have benefited from your collective wisdom and support, and I am immensely grateful for the positive impact you all have had on this thesis.

Let me begin by first thanking my adviser, Prof. Hiren Patel. I lucked out working with Hiren, and I consider myself even luckier to have been mentored by him for close to nine years (2012-2021)! I have benefited greatly from his supervision style that puts student training at the forefront. His training over these years have had a transformative effect on my technical thinking and skills. Hiren taught me how to understand and communicate technical challenges and observations at the right abstraction level that focuses on the essential details. In addition to my technical training, Hiren imparted several life lessons at crucial points in my PhD tenure that have shaped me into a better person. I am immensely grateful for his guidance and training, and look forward to many years of collaboration and friendship.

I am extremely grateful to the members of my thesis committee for their time and constructive feedback on my thesis: Prof. Jörg Henkel, Prof. Ken Salem, Prof. Rodolfo Pellizzoni, and Prof. Nachiket Kapre.

I feel extremely privileged and humbled to have interacted with some amazing instructors, academics, and students during my time at the UWaterloo. I sincerely thank you all for your wisdom and inspiration. Special thanks goes to Mohamed Hassan, with whom I wrote the first work that would lay the foundation for this thesis. Many thanks to the fantastic Zhuanhao Wu, with whom I had the pleasure of collaborating on a couple of works that are part of this thesis. I look forward to more collaborations with Zhuanhao. Special thanks to past and current members of the CAESR lab for their knowledge and support: Dan Wang, Zhuoran Yin, Yunling Cui, Nivedita Sritharan, Paulos Tegegn and Artem Klashtorny.

My time in Waterloo wouldn't have been nearly as rich without a wonderful set of friends: Hemant Saxena, Sharath Ibrahimpur, Jimit Mazmudar, Priya Soundararajan, Dhinakaran Vinayagamurthy, Abhinav Bommireddi, Retnika Devasher, Paulous Tegegn, Varuna Manivannan, and Karthik Velakur. From yearly camping trips, happy hours, and Friday night hangouts, you guys made Waterloo a special place for me and happily complimented the rigorous working hours of graduate school. I thank you all for your support and I consider myself blessed for your friendship. I want to especially thank Hemant for being a great friend and fellow companion on this PhD journey. Your levelheadedness and strong support helped me in this journey, and I cannot

thank you enough. Many thanks to Srinivas Suryanarayan, Navnit Narayan Das, Akshay Shrivastava, and Sanket Jaiswal for being wonderful friends over many years despite geographical distances and different time-zones.

Finally and especially, I am immensely grateful to my parents Mandakolathur V. Mohan and Revathy Mohan for their support and love during this education journey. It is not possible for me to fully express my gratitude for their pure and unconditional love and support. They instilled in me the value of seeking good education and have sacrificed many luxuries so that I may receive a good education in India and Canada. My parents have been sources of strength during difficult times and celebrated every success, no matter how small, with great pomp. They are the sole reason for my success, happiness, and making my dreams come true.

Dedication

This thesis is dedicated to my parents for their love, support, and wisdom thoughout my life. I would never have made it here without you.

Table of Contents

List of Figures

List of Tables

List of Publications

A large part of the contents of this thesis has been previously published in peer-reviewed conference and journal publications which I have co-authored. The use of the content, from the listed publications, in this thesis has been approved by all co-authors. For each publication, I present a list of my contributions.

1. Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel, "Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems" in IEEE Transactions on Computers (TC), 2020 [74]

    - Developed, designed and implemented the cache coherence mechanisms
    - Developed the timing analysis
    - Execution of empirical evaluation
    - Wrote portions of the article

2. Anirudh Mohan Kaushik and Hiren Patel, "A Systematic Approach to Achieving Tight Worst-Case Latency and High-Performance Under Predictable Cache Coherence" in IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2021 [76]

    - Deveopled, designed and implemented the cache coherence mechanisms
    - Developed the timing analysis
    - Execution of empirical evaluation
    - Wrote significant portion of the article

3. Anirudh Mohan Kaushik and Hiren Patel, "Automated Synthesis of Predictable and High-Performance Cache Coherence Protocols" in IEEE Design Automation and Test in Europe (DATE), 2021 [75]

    - Deveopled and designed the synthesis tool
    - Execution of empirical evaluation
    - Wrote significant portion of the article

4. Anirudh Mohan Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren Patel, "CARP: A Data Communication Mechanism for Multi-Core Mixed-Criticality Systems", in IEEE Real-Time Systems Symposium (RTSS), 2019 [77]

- Deveopled and designed the cache coherence mechanism and contributed to implementation
- Developed the timing analysis
- Execution of empirical evaluation
- Wrote significant portion of the article

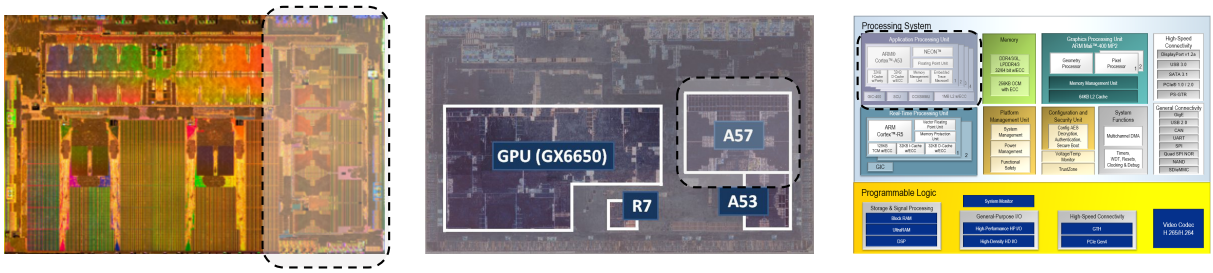
Chapter 1

Introduction

Today, computing systems are no longer limited to data-centers and supercomputers for delivering fast internet services and executing highly complex scientific computations respectively or desktop and laptop devices for personal use. In the last ten years, *embedded* computing systems that sense and interact with the physical world have become ubiquitous and pervasive in our daily lives. Examples of domains where embedded computing systems are used range from wearable and smart home technologies (smart watches, smart thermostats, video doorbells) to avionics, automotive, and robotic domains. Among these domains, computing systems used in avionics and automotive must not only compute correct results for correct functioning (logical correctness), but complete these computations within strict timing constraints (temporal correctness) for *correct and safe* operation. Failure to complete the computations within the timing constraints can result in dire catastrophic consequences such as loss of human lives. Such compute systems that depend on both logical correctness and temporal correctness are called *real-time systems*, and are the focus of this thesis.

As an example, consider an object avoidance application deployed in vehicles to realize semi/full autonomous driving capabilities [89]. This application detects the presence of objects on the current vehicle trajectory path, and informs the human driver of any detected objects. The human driver performs actions to safely steer the vehicle to avoid colliding with the detected object. Clearly, for safety of the human driver, this application must not only detect the object (functional correctness) but must do so in a timely manner (temporal correctness) so that the human driver can react to the detected object in a safe manner. Automatic braking system (ABS), airbag control systems, autopilot functions, and safety-critical radar applications are other examples of applications from the automotive and avionics domains that must satisfy functional and temporal correctness for safe and correct operation.

1.1 Multi-Core Real-Time Systems



(a) Tesla full-driving chip [140] (b) Renesas H3 MPSoC [139] (c) Xilinx Ultrascale+ MPSoC [17]

Figure 1.1: MPSoCs with multi-core computing units (highlighted) used in real-time systems.

There has been increasing attention and adoption of *multi-core computing platforms* to satisfy the demand for high computation capabilities in current real-time systems [28, 103, 106, 127, 144, 145]. For example, ARM projects a 100× increase in computing performance between 2015-2024 in the automotive domain to enable future advanced driver assistance systems (ADAS) based on radar and computer vision technologies [146]. Multi-core computing platforms consolidate multiple processing units (referred to as cores or CPUs) onto a single component resulting in enhanced application performance and reduced power consumption. Furthermore, the consolidation of multiple processing units offers advantages with respect to the size, weight, and power (SWaP) constraints imposed by the embedded environment in which real-time systems are deployed.

The current trend in industry is to use commercially available off-the-shelf (COTS) multi-core platforms in real-time systems. This trend in using COTS multi-core platforms in real-time systems is driven by the following reasons: (1) these platforms are readily available, (2) multi-core platform vendors spend considerable efforts in verifying their implementation before making them commercially available, and (3) availability of robust software ecosystems that enable designers to deploy their software applications with minimal effort. For these reasons, COTS multi-core platforms for real-time systems are preferred over custom in-house developed multi-core platforms as they shorten deployment time and minimize cost [12, 29, 148]. Embedded multi-processor system-on-chips (MPSoCs) developed for automotive and avionics domains such as Tesla's full self-driving computer [145], Renesas's R-Car SoC [144], Xilinx Ultrascale+ MPSoCs [18] feature COTS multi-core computing units alongside other compute accelerators such as as digital signal processing (DSP) units and graphics processing units (GPUs). We refer to real-time systems that use multi-core computing units as *multi-core real-time systems*. We refer the reader to [112] for a recent comprehensive survey of COTS multi-core computing units used in real-time systems.

The memory hierarchy of current multi-core platforms feature multiple levels of *hardware caches*, which are small on-chip memory structures that are placed close to the core's execution execution pipeline. These caches store a subset of data in the shared main-memory that is frequently used, and provide fast data access to cores due to their proximity to the cores. Some of these cache levels are private to each core and some caches levels are shared across all cores, and they are essential for average-case performance. The multi-core platforms referred to in this thesis are *cached-based* multi-core platforms. Since hardware caches are primary performance features in multi-core platforms, there is a rich body of research work devoted towards leveraging their performance benefits for real-time systems [15, 23, 79, 94, 115, 131, 150].

1.2 Timing Predictability of Multi-Core Real-Time Systems

Analyzing the temporal behavior of a real-time application prior to execution on the real-time system, which is referred to as *static timing analysis*, is crucial towards guaranteeing its temporal correctness. A real-time application comprises of multiple software routines or *tasks*, and each task must satisfy logical and temporal correctness. At a high level, this static timing analysis consists of two phases: (1) deriving an upper bound on the execution times of the tasks constituting the real-time application – *worst-case execution time (WCET) analysis* and (2) determining an execution schedule of the tasks such that the computed WCET of each task is not exceeded – *schedulability analysis*. Deriving the WCET of a task requires information about the task's software structure and the micro-architecture of the underlying compute platform. The derivation of the task's WCET must be *safe* in that it should be greater than or equal to any possible execution time of the task and *tight* in that it is the lowest WCET possible.

The micro-architecture of COTS multi-core platforms poses challenges to carry out precise WCET analysis resulting in the derivation of *loose* or *pessimistic* WCETs. The timing analysis challenges with multi-core platforms are primarily attributed to the presence of *shared hardware resources* that are shared between the multiple cores such as I/O interconnects, shared on-chip memories, and the shared main-memory [104, 116]. Multiple real-time tasks simultaneously executing on different cores *interfere* with each other for accesses to shared hardware resources thereby impacting their timing behavior and complicating the timing analysis. To put this into perspective, Nowotsch et al. [104] showed that a read operation to the shared main-memory (DRAM) on a multi-core platform used in real-time systems increased by more than 14$\times$ when the number of concurrent cores executing tasks increased from one core (41 cycles) to eight cores (604 cycles). As a result, computing the WCET of a real-time task on a multi-core platform must take into account the timing interference caused due to shared hardware resources, which results in *pessimistic* WCET estimates. The risk of using such pessimistic WCET estimates is

that the schedulability analysis may deem a real-time application unschedulable on a compute platform (in other words, no feasible execution schedule such that all tasks of the application satisfy temporal correctness) even though it is safe to execute the real-time application on the compute platform. This prevents real-time applications from being deployed, which limits the functionality of the real-time system.

To this end, there is a large body of research devoted towards making multi-core real-time systems *timing predictable* or simply predictable in order to enable the timing analysis to compute precise WCET. Predictability is defined in Definition 1.

Definition 1. *Predictability is a property of a system that makes it easy for timing analysis to compute WCET of real-time tasks and guarantees that all possible execution times of tasks deployed on the system are within their WCET [55].*

Prior works in this research can be classified into four categories:

- (C1) New theories and analyses methodologies that improve the precision of the timing analysis on multi-core real-time systems [7, 95, 96, 161, 165]
- (C2) Empirical analysis of COTS multi-core platforms to capture the timing impact of shared hardware resources on WCET [29, 41, 103, 116]
- (C3) Software techniques such as real-time operating system (RTOS) extensions and control of existing hardware features on multi-core platforms to limit timing interference [14, 23, 25, 47, 50, 52, 61, 79, 80, 84, 86, 99, 111, 141]
- (C4) Novel computer architecture components that are designed with predictability in mind [15, 56, 63, 64, 78, 86, 91, 109, 118, 128, 131, 150] [1].

Table 1.1 puts into perspective the impact of one recent prior work from each of these categories towards improving the predictability of multi-core real-time systems. We refer the reader to [40] for a comprehensive survey and critique of research efforts devoted towards addressing timing interference due to shared hardware resources in multi-core platforms.

[1] Note that prior works under C4 propose hardware features not available in COTS multi-core platforms. The desired outcome of these works is to convince COTS multi-core manufacturers and vendors to implement them in future COTS multi-core platforms.

Prior works	Category	Brief description	Impact
Mancuso et al. [96]	C1	A novel *WCET analysis theory* that reduces pessimism in the computed WCET of an application using knowledge about the application's execution in isolation (no interference)	Up to *60% reduction in the pessimism of estimated WCET* compared to state-of-the-art approaches
Radojković et al. [116]	C2	*Empirical analysis* of several COTS multi-core platforms to identify impact of shared hardware resource interference on application execution time	Identifies COTS multi-core platforms *suitable for real-time systems* and presents a methodology to identify computer architecture components in multi-core platforms that impact predictability
Ward et al. [153]	C3	A *software framework* that provides fine grained control of shared hardware resources (cache and DRAM partitioning, cache locking, cache scheduling) to *limit timing interference* between concurrent real-time tasks on shared hardware resources	Fine grained control *reduced* the timing interference on shared hardware resources resulting in scheduling tasks that were previously unschedulable due to pessimistic WCET estimates
Valsan et al. [150]	C4	New *predictable computer architecture components* in the form of per-core hardware control of outstanding memory requests to the shared memory to limit contention of shared hardware resources (miss status handling registers)	*Up to 19% reduction in the WCET* compared to classic cache partitioning techniques.

Table 1.1: Summary of prior research works on improving predictability of multi-core real-time systems.

1.3 Thesis Focus: Achieving Predictable and High-Performance Shared Data Communication Between Multiple Cores

A common assumption underlying many of these prior works is that the real-time applications deployed on the multi-core platforms consist of *independent tasks*. It is our take that this means that real-time tasks do *not* communicate data with each other. This assumption is no longer representative of current and emerging practical real-time systems [43, 60, 72]. Furthermore, as demands for more integrated functionalities in real-time systems continue to rise shared data communication between multiple simultaneously executing real-time applications on different cores will be necessary to realize such functionalities. Examples of applications deployed in real-time domains that feature data communication include machine learning algorithms for training and classification [31, 85, 89, 172] and diagnostic real-time tasks that communicate frequently changing values from sensors and engines [43, 60]. As a concrete example, in state-of-the-art autonomous driving systems, multiple machine learning tasks execute in parallel and operate on the data captured by various sensors (shared data) to perform real-time object detection and vehicle localization computations [89].

Shared data communication between real-time applications executing on different cores makes multi-core platforms even less amenable for timing analysis. This is because the timing behavior of a real-time application accessing shared data not only depends on the timing interference due to shared hardware resources but also on the *memory state* of the shared data due to past memory activity from other applications on the same shared data. As an example, consider the

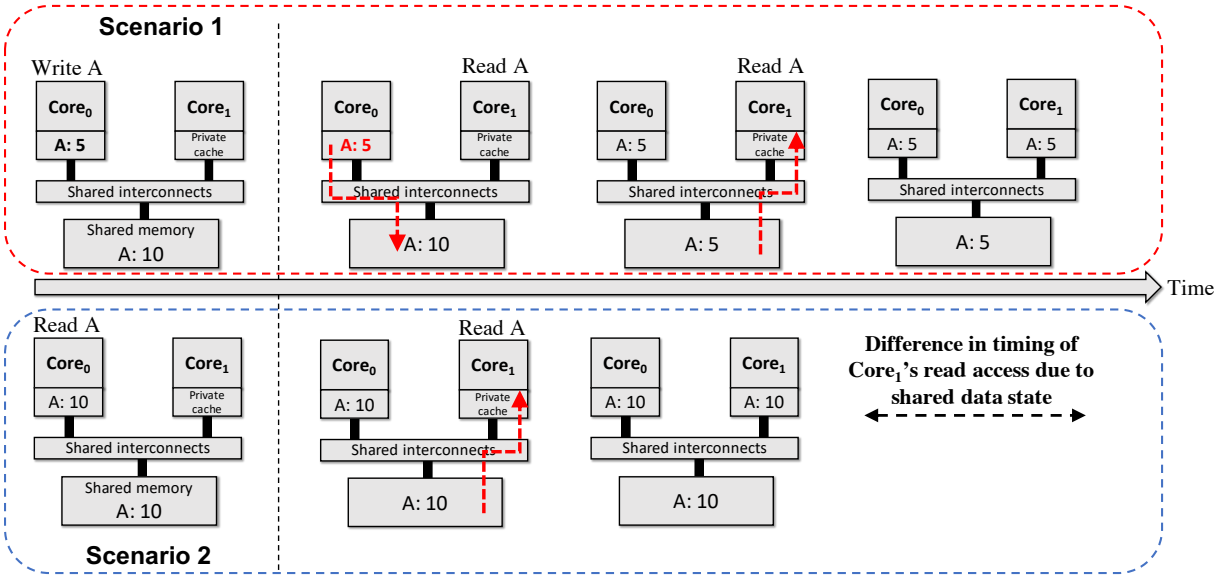


Figure 1.2: Illustrative example of impact of shared data accesses on timing behavior of real-time applications.

scenarios shown in Figure 1.2 where two cores (Core_0 and Core_1) are accessing shared data with memory address A. In the first scenario, Core_0 has modified A with a new value, and this is cached in Core_0's private cache. In the second scenario, Core_0 has an unmodified version of A in its private cache. Hence, the data state of A in Core_0's private cache is different in both scenarios. This results in different timing behavior of Core_1's access to A as shown. In the first scenario, Core_0 has to respond by updating the shared memory version of A so that Core_1 receives the most up-to-date version of A from the shared memory. On the other hand, in the second scenario, Core_1 receives the correct data from the shared memory without any involvement of Core_0. Therefore, precise timing analysis of shared data communication involves accounting for timing interference due to shared hardware resources on two fronts – (1) a core's own access (Core_1's read to A in both scenarios) and (2) other cores' responses based on the memory state of the shared data (Core_0's update to shared memory in the first scenario). To handle the timing analysis challenges associated with shared data communication, recent research have focused on designing *predictable shared data communication mechanisms* [13, 14, 25, 52, 60, 79, 86, 87, 153].

1.3.1 Motivation: Existing Predictable Shared Data Communication Mechanisms Constrain Application Performance

Designing for predictability entails designing components that are amenable to WCET analysis. As a result, designing for predictability is concerned with *worst-case scenarios* such that the computed WCET estimates are safe. On the other hand, designing for high-performance entails designing components that optimize for the *common case* or *average-case scenarios*. For example, speculative execution and deep cache hierarchies are high-performance micro-architectural optimizations common in COTS computing platforms that make computing WCET estimates difficult. Such architectural optimizations *threaten* predictability [11, 147]. As a result, predictability and high-performance are typically *conflicting* design goals where the former is concerned with optimizing worst-case scenarios and the latter is concerned with optimizing average-case scenarios [15, 24, 25].

Since predictability is a paramount consideration for real-time systems, state-of-the-art predictable shared data communication mechanisms trade away high-performance for enabling timing analysis. Examples of such mechanisms are:

- (M1) Private cache bypassing of shared data [13, 25, 86, 153],
- (M2) Co-locating real-time applications that communicate shared data to execute on the same core [14, 25, 52],
- (M3) software changes to real-time applications to eliminate shared data communication such as shared data duplication [25, 60, 79].

These mechanisms are adopted by industry as discussed in a recent survey on real-time systems industry practices [5]. To put this in context with the example in Figure 1.2, mechanisms in M1 prevent cores from caching A in their private caches, mechanisms in M2 force the application to execute on a single core, and mechanisms in M3 duplicate A into two versions where each version has a unique memory address. At a high level, these prior mechanisms place one of the following constraints: *(1) how data is cached in the cores' cache hierarchy and (2) when and where real-time applications are executed* to achieve predictable shared data communication. These data caching and application execution constraints in turn affect performance. Data caching constraints prevent cores from fully utilizing their private memory hierarchy (caches) that have been optimized for low latency access thereby incurring *high compute latency*. Application execution constraints prevents the usage of all available cores on the multi-core platform thereby *throttling compute throughput*. The consequences of such constraints (high compute latency and low compute throughput) goes *against* some of the key reasons for adopting multi-core platforms

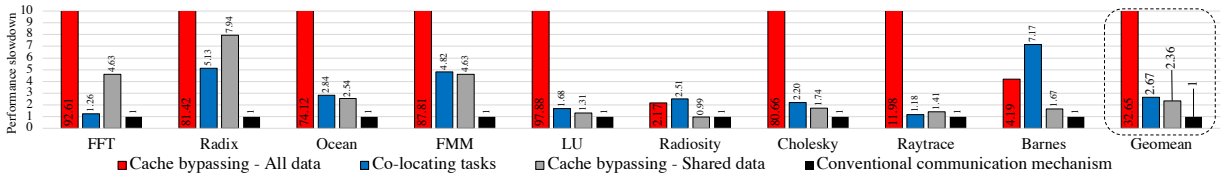


Figure 1.3: Performance slowdown of state-of-the-art approaches to predictable shared data communication against a conventional multi-core communication mechanism for the SPLASH-2 workloads on a 4-core multi-core platform. These results are taken from [74]. Lower is better.

(low compute latency and high compute throughput), and hence, prevent real-time systems that employ such predictable shared data communication mechanisms to take *full advantage* of the compute capabilities offered by multi-core platforms. Figure 1.3 highlights the performance slowdown of state-of-the-art predictable shared data communication mechanisms compared to the conventional multi-core communication mechanism deployed in multi-core platforms, which is hardware cache coherence [97]. A clear takeaway from Figure 1.3 is that state-of-the-art predictable shared data communication mechanisms exhibit up to 97× performance slowdown (2.3×-32× average slowdown), which underscores the conflicting design trade-offs between predictability and performance.

1.3.2 Design Dilemma: Reconciling Predictability and High-Performance

The conflicting design goals of predictability and high-performance puts real-time system designers and architects in a *dilemma* about the best way to reconcile them when designing a shared data communication mechanism. This dilemma is a recurring feature in real-time systems research, and features for different components of the real-time compute stack of which shared data communication mechanism is one component. Prior research works that addressed this dilemma for different components of the real-time compute stack espoused one of the following two design philosophies:

1. **Predictable Computer Architecture Design:** Research works that followed this design philosophy argued that the micro-architecture of compute platforms used in real-time systems must be designed with timing predictability as a first class design principle [15, 33, 56, 63, 65, 67, 78, 88, 91, 109, 118, 126, 128, 131, 150]. These works followed one of two approaches: (1) design the entire compute stack (micro-architecture, instruction set architecture, and associated compiler framework) from ground up with predictability in mind [33, 88, 109, 126, 128] (*complete approach*) or (2) analyze the predictability guarantees of a particular computer architecture component such as caches and memory con-

trollers present in COTS multi-core platforms and design extensions to improve their predictability [15, 56, 63, 65, 67, 78, 91, 131, 150] (*compositional approach*).

2. **Predictable Software Runtime Design:** Research works that followed this design philosophy argued that COTS multi-core platforms will continue to exhibit low predictability as compute platform vendors will most likely optimize for high-performance due to market factors. Therefore, it is the responsibility of the software (application or RTOS) to guarantee predictability [14, 23, 25, 47, 50, 52, 61, 79, 80, 84, 86, 99, 111, 141].

State-of-the-art predictable shared data communication mechanisms described in the previous section (Section 1.3.1) espoused the second design philosophy. In this thesis, we adopt the first design philosophy and propose *hardware-based* shared data communication mechanisms to achieve predictable and high-performance shared data communication. Therefore, the research contributions of this thesis fall under the *predictable computer architecture design philosophy*. Our motivation for choosing this design philosophy is based on the observation that software approaches to predictable shared data communication such as cache bypassing and task mapping have reached a *ceiling* on their performance returns. This is because, these approaches have *limited visibility* of the underlying micro-architecture and hence, wield *limited control* on the performance and predictability trade-offs. As a result, it is highly unlikely that newer communication mechanisms that follow the same design philosophy will observe significant performance returns above this performance ceiling.

On the other hand, predictable computer architecture design allows for *finer control* of the predictability and performance trade-offs through micro-architectural changes. Hence, a shared data communication mechanism designed by making *performance-preserving predictable* micro-architectural changes to an existing hardware-based shared data communication mechanism can achieve better performance compared to state-of-the-art approaches while guaranteeing predictability. This leads to the following research question that this thesis is concerned with:

Thesis question: Is it possible to design a hardware-based predictable shared data communication mechanism such that its average-case performance guarantees go beyond the performance ceiling of state-of-the-art predictable shared data communication mechanisms?

The approach taken in this thesis proposes micro-architectural changes to *hardware cache coherence mechanisms*, which are existing data communication mechanisms in COTS multi-core platforms. The research contributions presented in this thesis answer the above proposed research question in the affirmative.

1.3.3 Our Approach: Predictable and High-Performance Shared Data Communication through *Hardware Cache Coherence*

Hardware cache coherence is a staple high-performance feature in multi-core platforms that facilitates correct shared data communication between multiple cores [97]. At a high level, hardware cache coherence enables multiple cores to *correctly* access the most up-to-date shared data, and allow multiple cores to simultaneously cache shared data in their private caches. However, hardware cache coherence has been overlooked as a potential shared data communication mechanism in real-time multi-core platforms. In this thesis, we take the first steps towards making hardware cache coherence mechanisms viable shared data communication mechanisms for multi-core real-time systems. We describe the design of multiple *predictable* hardware cache coherence mechanisms that are amenable to timing analyses for different real-time system models (Chapters 3 and 6), and discuss design techniques that effectively balance high average-case application performance and tight predictability requirements (Chapter 4). This thesis makes a strong case for the adoption and deployment of predictable hardware cache coherence mechanisms in real-time multi-core platforms by showing that they offer significant average-case performance benefits compared to existing predictable shared data communication mechanisms while still guaranteeing predictability. Hence, this thesis provides evidence for the following thesis statement:

Thesis statement: A shared data communication mechanism for real-time multi-core platforms that uses predictable hardware cache coherence delivers high average-case performance compared to state-of-the-art approaches while guaranteeing timing predictability.

1.4 Key Benefits of Proposed Approach

There are two key benefits of our proposed approach (predictable hardware cache coherence) that make it a compelling data communication mechanism for deployment in multi-core real-time systems.

1. **High average-case performance:** The research contributions in this work show compelling evidence that predictable hardware cache coherence offer better average-case performance compared to state-of-the-art predictable communication mechanisms. Figure 1.4 extends the results presented in Figure 1.3 with the performance slowdown of one of our approaches. Compared to the conventional hardware cache coherence mechanism, which is optimized for performance, our predictable hardware cache coherence mechanism exhibits 1.46× average performance slowdown and up to 2× performance slowdown (Radix benchmark). This performance slowdown is a consequence of guaranteeing predictability.

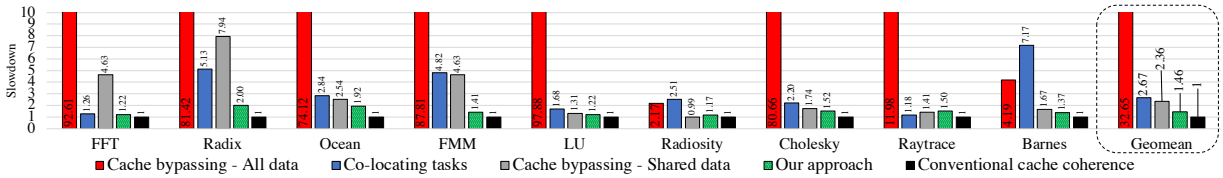


Figure 1.4: Performance slowdown of state-of-the-art approaches and one of our approaches to predictable shared data communication against a conventional hardware cache coherence mechanism for the SPLASH-2 workloads on a 4-core multi-core platform. These results are taken from [74]. Lower is better.

However, the performance slowdown of our approach is the *least* compared to state-of-the-art approaches. In fact, our approach achieves up to 4× improvement in performance compared to cache bypassing for shared data, which is the best among the state-of-the-art approaches. The key reason for this high-performance benefit is that unlike state-of-the-art approaches, predictable hardware cache coherence mechanisms do not enforce any data caching or application execution constraints. As a result, these mechanisms take better advantage of the computation capabilities offered by multi-core platforms compared to existing state-of-the-art approaches. As real-time applications continue to feature *high compute demands and frequent data communication* [72, 102, 146], the communication mechanisms proposed in the thesis are *well-positioned* to satisfy these demands without compromising on the critical timing predictability requirements.

2. **Better software design productivity:** Hardware cache coherence mechanisms handle correct shared data communication between cores transparent to the software application [97, 138]. This means that a software application designer does not need to worry about handling correct shared data communication in the software, and instead, can solely focus on exploring and implementing new software functionalities. This property of hardware cache coherence mechanisms holds for the predictable hardware cache coherence mechanisms proposed in this thesis. Real-time software designers design software applications that must meet functional and timing predictability guarantees. Predictable hardware cache coherence mechanisms can reduce some of the design burden by guaranteeing predictability of data communication. This improves the *design productivity* of real-time software designers as they can focus more on the functional implementation of the software application. With growing emphasis of using machine learning algorithms in real-time systems (for example, object detection and classification in self/fully driving autonomous vehicles [89]), high designer productivity of software application designers is crucial in order to implement advanced functionalities in real-time systems.

1.5 Thesis Contributions

This thesis makes the following contributions:

1. **Design Invariants for Designing Predictable Hardware Cache Coherence Mechanisms [74] (Chapter 3)**: The starting point of the research presented in this thesis begins with understanding *how* to build a predictable hardware cache coherence mechanism. We first comprehensively analyze all *sources of timing interference* that can arise when cores simultaneously cache correct versions of shared data in their private caches. We then propose a set of *design invariants*, which are general design guidelines, to design predictable cache coherence mechanisms. These design invariants do not impose any implementation constraints, and are not tied to any existing cache coherence mechanism implementation. We design three predictable cache coherence mechanisms using these design invariants. We perform timing analysis to compute the worst-case memory access latency under the proposed predictable cache coherence mechanisms, and empirically show that this derived worst-case memory access latency is *safe and tight*.

2. **Balancing Predictability and High-Performance Guarantees in Predictable Hardware Cache Coherence Mechanisms [76] (Chapter 4)**: A shared data communication mechanism that is predictable but has large worst-case memory access latency is an *inferior* choice compared to another predictable communication mechanism with lower worst-case memory access latency irrespective of the former's performance benefits over the latter. This is because worst-case timing estimates are used in the schedulability analysis (Section 1.2). To this end, the second contribution presents a systematic analysis framework to *understand* the relationship between design features of a predictable cache coherence mechanism and the resulting worst-case memory access latency. Using this framework, we present a technique to *lower* the worst-case memory access latency under predictable cache coherence mechanisms while *preserving* their performance benefits over state-of-the-art communication mechanisms.

3. **Automating the Construction of Predictable and High-Performance Cache Coherence Protocols [75] (Chapter 5)**: In hardware cache coherence mechanisms, the cache coherence protocol is the main component that enforces rules for correct shared data communication between cores. Designing a predictable and high-performance cache coherence protocol is non-trivial as it requires accounting for different types of memory communication scenarios: (1) accesses to shared hardware resources such that they are managed in a predictable manner and (2) simultaneous memory interleaving from multiple cores to the same shared data such that they are handled in a non-stalling manner. Accounting for

these scenarios adds to the design complexity of predictable and high-performance cache coherence protocols. To manage this design complexity, the third contribution proposes SYNTHIA, a tool that *automates* the construction of predictable and high-performance cache coherence protocols. The input to this tool is a simple specification of the cache coherence protocol that is devoid of any predictability and performance guarantees. The tool refines this input and adds details that guarantee predictability and high-performance.

4. **Designing Criticality Aware Predictable Cache Coherence Mechanisms for Mixed Criticality Systems [77] (Chapter 6)**: A mixed-criticality system (MCS) is a real-time system where tasks running on the system have different safety requirements (timing and average-case performance requirements). To support shared data communication between tasks of various criticality levels, the underlying communication mechanism must ensure that the timing behavior of high critical tasks are *not* impacted by low critical tasks. The fourth contribution presents CARP, a *criticality aware* predictable hardware cache coherence mechanism that allow tasks of different criticality levels to communicate data with each other while ensuring that shared data communication to/from low critical tasks do not impact the timing behavior of high critical tasks.

1.6 Structure of Thesis

Chapter 2 provides a background on multi-core platforms and hardware cache coherence mechanisms and presents a brief overview of related works. Chapters 3-6 are the main research contributions of this thesis. Chapter 3 presents a design template that guides the design of predictable hardware cache coherence mechanisms for hard real-time systems and describes the construction of three predictable hardware cache coherence mechanisms. Chapter 4 presents a systematic formal analysis framework to capture the relationship between the design of predictable hardware cache coherence mechanisms and their timing analysis. This framework guides the design of better predictable hardware cache coherence mechanisms that result in lower worst-case latency while maintaining their average-case performance benefits. Chapter 5 describes SYNTHIA, a tool that automates construction of predictable hardware cache coherence protocols. SYNTHIA is designed using the formal analysis framework developed in Chapter 4. Chapter 6 describes the design of predictable hardware cache coherence mechanisms for mixed-criticality systems (MCS), which are real-time systems that deploy tasks of varying timing criticality levels. Chapter 7 shows how our research contributions can be integrated with other prior works to realize end-to-end timing predictable and high-performance real-time multi-core systems. We conclude this thesis with Chapter 8 that lists some areas of future works based on our research contributions.

Chapter 2

Background and Related Works

This chapter provides necessary background on multi-core platforms and hardware cache coherence mechanisms that allow for coherent shared data communication between multiple cores on multi-core platforms and a brief overview of related works. We discuss specific related works in detail based on the research contributions in Chapters 3-6.

2.1 Multi-Core Platforms

Figure 2.1 shows the typical components of a multi-core platform. A multi-core platform consolidates several processing units called *cores* onto a single silicon die. Each core consists of parallel circuitry for decoding and executing a sequence of instructions and dedicated register files that store data manipulated and operated on by the instructions. A core fetches the data required by the instructions into the register files from the *memory hierarchy*, which is a hierarchy of memory components with different data capacities and access latency. The focus of this thesis is on the data movement in the memory hierarchy. The memory hierarchy includes a hierarchy of *caches*, which are small low-latency SRAM-based memory components and a high-latency DRAM-based main-memory that stores all data; the cache hierarchy stores a subset of data in the main-memory. Table 2.1 describes the data capacity and access latency of different memory components in a typical multi-core platform. Both instructions and data share the same physical memory space (Von Neumann architectures). Some levels of the memory hierarchy are *private* to each core and other levels are *shared* across all cores. For example, in most multi-core platforms each core has one to two levels of private cache memories, which are referred to as the level one (L1) and level two (L2) cache levels. These private cache memories are small and are placed

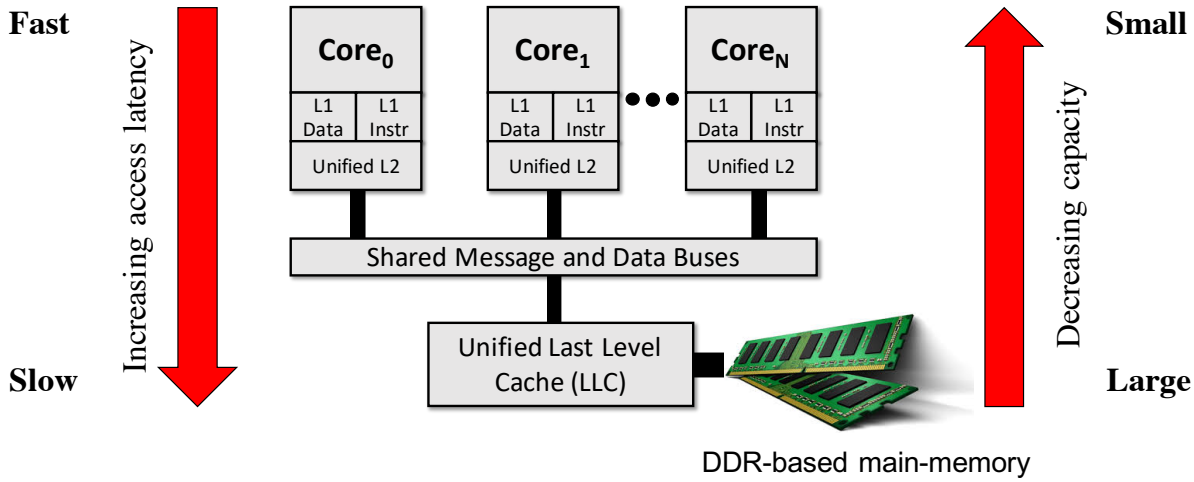


Figure 2.1: Typical multi-core platform. Capacity and access latency trends of memory components in memory hierarchy highlighted.

Component	Capacity	Access latency
Register files	1KB	1 cycle
Private L1 cache	32KB	4 cycles
Private L2 cache	512KB	10 cycles
Shared LLC	6MB	38 cycles
Shared main-memory	4-12GB	65-100 cycles

Table 2.1: Capacity and access latency of different memory components in memory hierarchy of Intel's Xeon 5500 processors (Nehalem-EP) multi-core processors. Data taken from [57].

close to the core's instruction execution circuitry for fast access. Multiple cores share the larger cache levels (last level caches) and the DRAM-based main-memory. Cores access these shared memory components through a shared interconnect.

In this thesis, we consider multi-core platforms that have a 3-level memory hierarchy with private split L1 caches for instruction and data, a unified shared last level cache (LLC), and a DRAM-based main-memory. The memory hierarchy is *inclusive* in that the L1 caches store a subset of the data present in the L2 cache, and the L2 cache stores a subset of data in the main-memory. Cores access the shared LLC and main-memory through a bus interconnect. The exact parameters of the multi-core model assumed by the latency analysis and evaluation are specific to the research contribution, and are described in the appropriate chapters.

Note that this thesis focuses on improving the timing predictability of one component of the multi-core compute stack, which is the data communication component, through predictable

hardware cache coherence mechanisms. This data communication component encompasses the private and shared levels of the memory hierarchy. The following chapters derive the *worst-case latency* (WCL) of a memory request under the proposed predictable hardware cache coherence mechanisms. This WCL is a building block towards computing the total WCET of a task. We adopt a *compositional timing analysis* approach [59] to use the derived WCL of a memory request to compute the total WCET of a task. At a high level, compositional timing analysis divides a multi-core platform into independent components and the timing contributions of each component can be combined to compute the total WCET of task. In Chapter 7, we discuss in detail the WCET computation using our proposed predictable hardware cache coherence mechanisms using compositional timing analysis.

2.2 Hardware Cache Coherence

The objective of a cache coherence mechanism is to enable cores to cache the most recent write on shared data. These mechanisms are the primary data communication mechanisms in existing multi-core platforms [97].

Illustrative example of incoherent data sharing. Incoherent sharing of data occurs when multiple cores read different versions of the same data that is present in their private cache hierarchies. Figure 2.2 shows a scenario where cores share incoherent data. Consider the shared data in address A. Initially, both cores (Core_0 and Core_1) do not have A in their private caches. At ❶, Core_0 performs a write operation to A. Since Core_0 does not have A in its private cache, this write operation is a cache miss. Core_0 communicates the write operation on the interconnect, and the shared memory supplies the value of A (❷), which is then updated by Core_0. At ❸, Core_1 performs a read operation to A, and this read operation is a cache miss. Core_1 communicates its read operation on the interconnect. The lack of a cache coherence mechanism can cause the shared memory to send an *outdated* value of A. This is shown in ❹ where Core_1 receives an outdated value of A resulting in incoherent view of A across the cores.

Hardware cache coherence. A cache coherence mechanism avoids data incoherence by deploying a set of rules to ensure that cores access and cache the correct version of data at all times. The main component in a hardware cache coherence mechanism is a *cache coherence protocol*, which is a state machine that deploys the set of rules. Typically, the coherence protocol maintains data coherence at cache line granularity, which is a fixed size collection of data. The protocol state machine implements these rules with a set of *coherence states* that convey access permissions (read, write) and other information about the cache line data, and *coherence state transitions* between states are triggered based on the memory activity of cores on the shared data. Figure 2.3 shows the implementation of a hardware cache coherence mechanism. The cores' cache

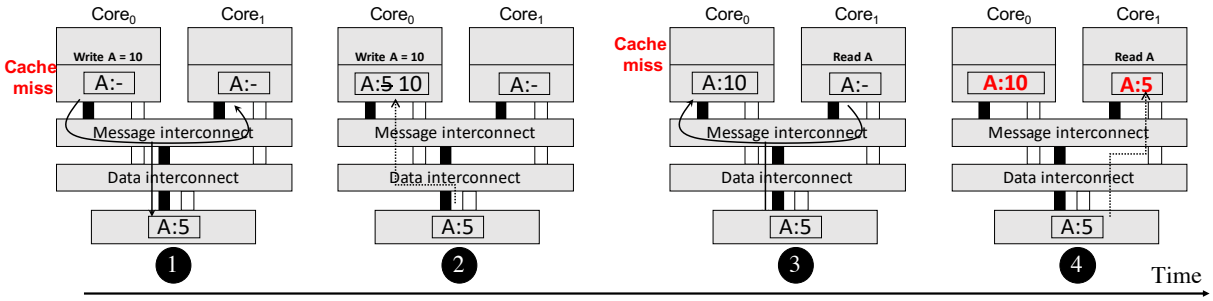
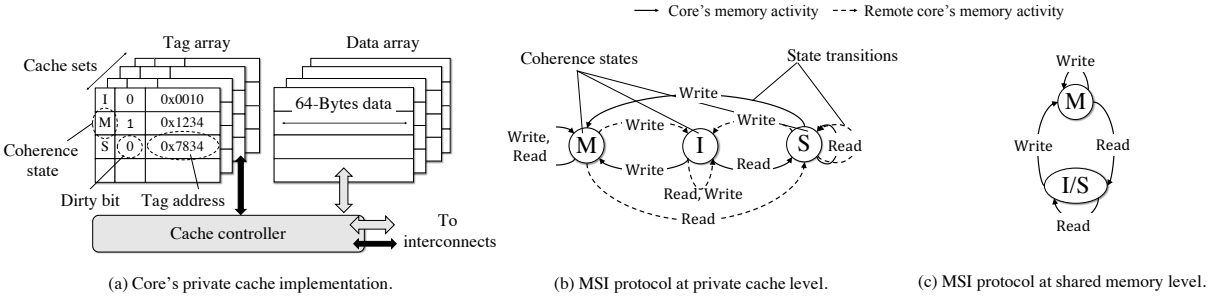


Figure 2.2: Example of incoherent shared data communication.



(a) Core's private cache implementation. (b) MSI protocol at private cache level. (c) MSI protocol at shared memory level.

Figure 2.3: Hardware cache coherence implementation of MSI cache coherence protocol.

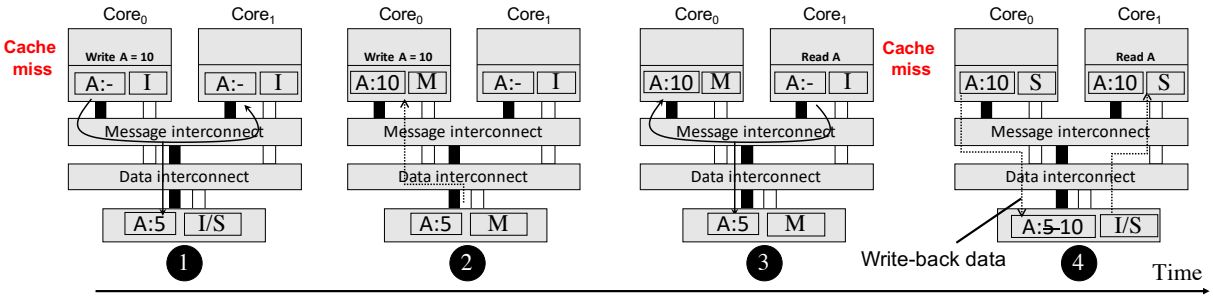
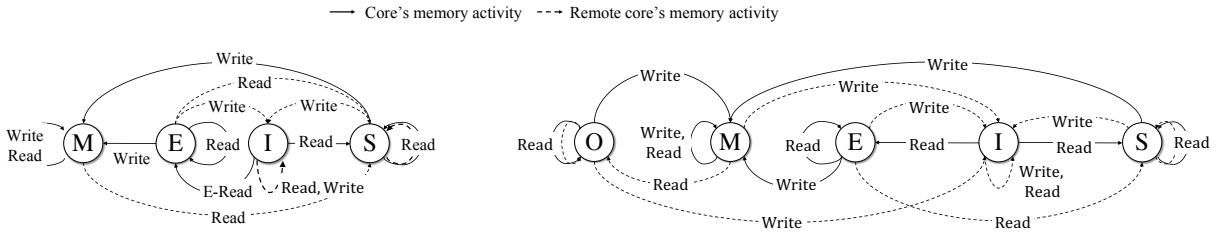


Figure 2.4: Coherent data sharing with MSI protocol.

controllers implement the coherence protocol, and the coherence states for the cache lines are maintained in the tag arrays.

Cache coherence protocols deployed in current multi-core platforms consist of three fundamental stable states, which establish the Modified-Shared-Invalid (MSI) protocol: *modified* (**M**), *shared* (**S**), and *invalid* (**I**) [138]. Figure 2.3 shows the MSI protocol state machine at the private cache and shared memory levels. A cache line in **M** means that the current core has written to it and it did not propagate the updated data to the shared memory yet. Only one core can have a specific cache line in a modified state, and is referred to as the *owner*. A core that has a cache line in **M** and observes remote memory activity on the cache line must update the cache line copy



(a) Modified-Exclusive-Shared-Invalid (MESI) protocol.

(b) Modified-Owned-Exclusive-Shared-Invalid (MOESI) protocol.

Figure 2.5: MESI and MOESI cache coherence protocol state machines at the private cache level.

in the shared memory. This operation is called *write-back* to shared memory. A cache line in **S** means that the core has a valid, yet unmodified version of that line. One or more cores can have versions of the same cache line in shared state to allow for fast read accesses. Cores that have the same cache line in the shared state are referred to as *sharers*. This constraint of one owner for a cache line or multiple cores sharing a cache line is referred to as the *single-writer multiple-reader* (SWMR) invariant [138]. A cache line in **I** denotes the unavailability of that line in the cache or that its data is outdated and no longer valid. At the shared memory level, the **I** and **S** are fused into one state **I/S**. This state means that the cache line data contents are not modified by any core. The shared memory state **M** means that there is a core that has updated the data contents of the cache line, and the data of the cache line in the shared memory is no longer up-to-date.

Figure 2.4 applies the MSI protocol to the example in Figure 2.2. Initially, both cores have A in **I** state (❶), and the shared memory has A in **I/S**. Core_0's write operation changes the coherence state of A in Core_0's private cache to **M** state (❷). The shared memory state of A correspondingly transitions to the **M** state. At ❸, Core_1 communicates its read operation to Core_0 and the shared memory. The shared memory cannot respond with data to Core_1 as it does not have the up-to-date version of A. Core_1's read operation causes Core_0 to perform a write-back of the updated data contents of A to the shared memory. After the data write-back, the shared memory sends the updated A to Core_1. At ❹, both cores have up-to-date copies of A, and coherence state of A in the private caches of Core_0 and Core_1 is **S**.

The Modified-Exclusive-Shared-Invalid (MESI) and Modified-Owned-Exclusive-Shared-Invalid (MOESI) protocols are cache coherence coherence protocols that apply performance optimizations to the MSI protocol, and these protocols are deployed in current multi-core platforms [28, 106]. These protocols are shown in Figure 2.5a and 2.5b respectively. The MESI protocol introduces the exclusive (**E**) as shown in Figure 2.5a. A core that receives a line in **E** state from the shared memory (E-Read in Figure 2.5a) guarantees that no other core has the same line in a valid state (**S** or **M** states). To enable this optimization, the shared memory keeps track of additional states to identify (1) a read-only copy of the line present in cores' private cache (**S**

state), (2) no copy of the line present in any cores' private cache (`I` state), and (3) an exclusive or modified copy of the line is present in a core's private cache (`E`/`M` state). The `E` state allows for one performance optimization for store requests. A core that has a line in the `E` state can complete a store request without issuing any coherence messages on the bus. This is because a core that has a line in the `E` state implies that there are no other cores that have the same line in their private caches. Hence, no private copies of the line need to be invalidated. We refer to this performance optimization as *silent stores*. On the other hand, in MSI protocol, a core performing a store operation on a line in `S` or `I` state must issue coherence messages on the shared bus before it can complete its store operation. The MOESI protocol in Figure 2.5b adds an optimization to the MESI protocol in the form of the owned (`O`) state, which minimizes data write-backs to the shared memory. In the MSI and MESI protocols, a core that has a cache line in `M` and observes a remote read operation must write-back the updated cache line data contents to the shared memory and changes the coherence state of the cache line to `S`. The `O` state in MOESI protocol optimizes this behavior by eliminating the write-back to shared memory; a core with a cache line in `M` state sends data to the requesting core on a remote read request, and transitions to the `O` state.

Types of hardware cache coherence. Cores' cache controllers and the shared memory change the state of their cache line copies based on the observed memory activity. There are two types of hardware cache coherence mechanisms based on how cores and the shared memory *observe* this memory activity: (1) *snooping bus-based* cache coherence and (2) *directory based* cache coherence [138]. In snooping bus-based mechanisms, cores *broadcast* their memory activity on a shared bus, and cores and the shared memory observe memory activity on a cache line by *snooping* this shared bus. Hence, the snooping bus is the *ordering point* for all memory activity. On the other hand, under directory based cache coherence mechanisms, cores communicate their memory activity to a centralized directory (unicast communication) that tracks information about a cache line across cores. The directory then communicates a core's memory activity to a cache line to other cores that have the cache line (multicast communication). Hence, the directory is the *ordering point* for all memory activity. The snooping bus-based coherence mechanisms is typically used for multi-core platforms with small core count (4-16 cores) due to bus scalability limits [138]. This dissertation focuses on snooping bus-based cache coherence mechanisms as they are typically implemented in multi-core platforms with a small number of cores, which is the case in current real-time systems [28, 106].

Memory activity under snooping bus-based cache coherence. We will use an example to describe a core's memory activity to a cache line under a snooping bus-based cache coherence mechanism. Consider the following scenario: core c_0 **issues** a read to cache line Z. The read to Z first checks c_0's private cache for the data contents of Z. On a private cache hit, the necessary data is supplied, and the read is marked complete. Private cache hits do not generate coherence activity. On a cache miss, the private cache controller of c_0 **generates** a coherence message of

the form Get(A). If a core issues a read request to Z, its cache controller generates a GetS(Z) message; GetS(Z) means to obtain a shared copy of Z. If a core issues a store request to Z, its cache controller generates a GetM(Z) message; GetM(Z) means to obtain a modified copy of Z. If Z is marked for eviction, and has been modified, the core first generates a PutM(Z) message, and then writes back the updated data contents of Z to the shared memory. If the core has Z in a shared state and wants to modify it, it generates an Upg(Z) message. The cache controller then **broadcasts** the GetS(Z)/GetM(Z)/PutM(Z)/Upg(Z) coherence message on the snooping bus. A coherence message is said to be **ordered** on the bus when all cores and the shared memory observe the corresponding memory request on the bus. A core observes its own messages on the bus (Own) as well as messages by other cores (Other). For example, c_0's GetS(Z) coherence message is observed by c_0 as OwnGetS(Z), and the other cores observes the same coherence message as OtherGetS(Z). Based on the cache coherence protocol transition, the coherence state of Z copies in c_0 and other cores change or remain the same.

Tables 2.2 and 2.3 show the complete *snooping bus-based* MSI cache coherence protocol state machines at the private cache level and shared memory level respectively. The state machines in Table 2.2 and 2.3 have more details specific to the snooping bus implementation that that described in Figure 2.3. The state machines have **`stable`** coherence states (s-states) and **`transient`** coherence states (t-states). Memory activity on a cache line start and end on s-states, and t-states capture pending memory activity information on the cache line. There are three types of t-states: (1) t-states that denote that a core is waiting for the corresponding coherence message to be ordered on the snooping bus (states suffixed with **_A**), (2) t-states that denote that a core is waiting for the requested data response (states suffixed with **_D**), and (3) t-states that are waiting for both the coherence messages to be ordered and data responses (states suffixed with **_AD**). As an example, consider a core that performs a read operation on a cache line that it does not have in its private cache under the MSI protocol. The starting stable state of the cache line is **`I`**. The core issues a GetS() coherence message and changes the coherence state of the cache line to t-state **`IS_AD`**, which denotes that a core has issued a GetS() coherence message and is waiting for the GetS() message to be ordered on the bus and the corresponding data response. When the coherence message is ordered on the snooping bus, the core changes the coherence state to t-state **`IS_D`**, which denotes that a core has observed its ordered coherence message, and is waiting for the data response. On receiving the data response, the cache line coherence state transitions to the stable coherence state **`S`**, and the core completes the read memory operation.

In addition to capturing pending memory state information, t-states also capture information regarding coherence state changes due to *interleaving* memory activity from other cores to the same cache line. Interleaving memory activity from other cores to the same cache line is possible due to the *non-atomic* implementation of the snooping bus [138]. Non-atomic snoop-

ing bus is preferred and implemented in existing multi-core platforms due to their performance benefits [138]. There are two ways to deal with interleaving memory activity on the same cache line. In the first approach, a core that has a cache line in a t-state can *stall* any coherence state changes until it completes its pending memory operation. However, such a protocol design will have poor performance as it introduces stalls. The second approach is a non-stalling approach wherein t-states can capture the impact of the interleaving memory activity to a cache line on a core's pending memory operation on the same cache line. We will illustrate this information capture using the following example. Consider c_0 has a cache line in t-state **IS_D**. c_0, all other cores, and the shared memory have observed c_0's read coherence message, and c_0 is waiting for the data response. While c_0 is waiting for the data response, another core c_1 broadcasts a GetM() coherence message to the same cache line. Since c_1 is performing a write operation, c_0 must invalidate the cache line on receiving the data response to maintain the SWMR invariant. As a result, c_0 moves the cache line from t-state **IS_D** to t-state **IS_DI**. Transient state **IS_DI** denotes that a core is waiting for the requested cache line data contents, and on receiving the cache line completes the read operation and moves to the **I** state. Note that this approach introduces additional t-states to eliminate stalling, and hence, offers better performance compared to the first stalling based approach.

In Table 2.3, the shared memory fuses the s-states **I** and **S** into one s-state **IoS**, which means that the data contents of the cache line are unmodified. This means that either some cores or no core have the cache line in their private caches. The t-states **IoS_D** denotes that the shared memory is waiting for a core to send updated cache line data contents to the shared memory (write-back) due to a cache line replacement or in response to another core's request. The shared memory on receiving the cache line data contents updates the shared memory copy and transitions to s-state **IoS**.

	Core events			Bus events						
States	Read	Write	Replacement	OwnGetS	OwnGetM	OwnData	OwnPutM	OtherGetS	OtherGetM	OtherPutM
I	Issue GetS()/IS_AD	Issue GetM()/IM_AD						—	—	—
S	Hit, Complete read	Issue GetM()/SM_AD	-/I					—	I	
M	Hit, Complete read	Hit, Complete write	Issue PutM()/MI_A					Write-back data, Send data/S	Send data/I	
IS_AD				-/IS_D		-/IS_A		—	—	—
IS_D						Complete read/S		—	-/IS_DI	—
IS_A				Complete read/ S				—	—	—
IM_AD					-/IM_D			—	—	—
IM_D						Complete write/M		-/IM_DS	-/IM_DI	—
SM_AD	Hit, Complete read/-		Stall		-/SM_D	-/SM_A			-/IM_AD	
SM_D	Hit, Complete read/-		Stall			Complete write/M		-/SM_DS	-/SM_DI	—
SM_A	Hit, Complete read/-		Stall		Complete write/M			—	-/IM_A	—
SM_DS	Hit, Complete read/-		Stall			Complete write, send data, write-back data/S		—	-/SM_DSI	—
SM_DI	Hit, Complete read/-		Stall			Complete write, send data, write-back data/I		—	—	—
SM_DSI	Hit, Complete read/-		Stall			Complete write, send data, write-back data/I		—	—	—
MI_A	Hit, Complete read	Hit, Complete write					Write-back data/I	Send data, write-back data/II_A	Send data/II_A	
IM_DI						Complete write, Send data/I		—	—	—
IS_DI						Complete read/I		—	—	—
IM_DS						Complete write, write-back data, send data/S		—	-/IM_DSI	—
IM_DSI						Complete write, Send data/I		—	—	—
II_A							-/I	—	—	—

Table 2.2: Private memory states for snooping bus-based MSI protocol. *issue msg/state* means the core issues the message *msg* and move to state *state*. A core issues a *read/write* request. Once the cache line is available, the core *reads/writes* it. A replacement triggers a cache line eviction. Highlighted cells denote impossible scenarios, and cells marked with '—' denote no change in state.

State	GetS	GetM	PutM	Data from core
IoS	Send data to requesting core/IoS	Send data to requesting core/M	—	
M	Clear owner/IoS_D	Update owner to requesting core/M	Clear owner/IoS_D	Write to memory/IoS_A
IoS_D	Stall	Stall	Stall	Write memory/IoS
IoS_A	Clear owner/IoS	—	Clear ownerIoS	

Table 2.3: Shared memory states for snooping bus-based MSI protocol.

2.3 Related works

This thesis proposes predictable and high-performance shared data communication mechanisms through hardware cache coherence mechanisms. We briefly discuss related works on predictable management of shared hardware resources and predictable shared data communication mechanisms in real-time multi-core platforms, and hardware cache coherence.

2.3.1 Predictable management of shared hardware resources

Timing interference caused due to shared hardware resources such as caches, main-memory, and interconnects compromises on the predictability of real-time multi-core platforms. As a result, there is a large body of research that focus on predictable management of such hardware resources [14, 15, 23, 25, 47, 56, 61, 63, 64, 66, 78, 80, 86, 91, 99, 109, 111, 118, 128, 131, 141, 150]. At a high-level, these techniques enforce *spatial or temporal isolation* to minimize timing interference due to simultaneous accesses to shared hardware resources from multiple cores. Spatial isolation techniques such as set-based partitioning and way-based partitioning in shared caches and DRAM bank partitioning across cores ensure that multiple tasks running on different cores access non-conflicting memory partitions [14,15,23,25,61,80,86,86,99,131,141,150]. Temporal isolation techniques space the execution of tasks across time such that multiple tasks running on different cores do not simultaneously conflict on shared hardware resources [47,63,66,109,111].

The research contributions of this thesis focus on predictable shared data communication between tasks running on different cores. This data communication happens through the memory hierarchy, which comprises of private and shared memory components. While many of these prior works assume that tasks running on cores do not communicate with each other, they are key ingredients along with our research contributions to design *end-to-end predictable and high-performance real-time compute platforms*. In particular, prior works that focus on predictable management of shared memory hierarchy components such as shared caches and shared interconnects *coupled* with predictable hardware cache coherence mechanisms enable predictable and high-performance memory hierarchies in real-time multi-core platforms. For example, in Chapters 3 and 6, we deployed our predictable shared data communication mechanisms on a shared bus that implements a time division multiplexing (TDM) arbitration policy. TDM arbitration achieves temporal isolation for cores' accesses to shared memory by dividing access time to the shared memory into fixed time slots that are allocated to cores. A core has exclusive access to the shared memory in its allocated time slot, thus achieving temporal isolation.

2.3.2 Predictable shared data communication mechanisms

As discussed in Section 6.2, state-of-the-art approaches place data caching and application execution constraints to achieve predictable shared data communication between cores [13, 14, 25, 52, 60, 79, 86, 153]. At a high level, these constraints prevent multiple cores from *simultaneously* caching multiple copies of the same shared data in their private caches. For example, approaches that enforce private cache bypassing of shared data [13, 25, 86, 153] prevent cores from caching shared data in their private caches. As a result, such approaches do not fully utilize a core's private memory hierarchy (private caches) resulting in reduced application performance. Consider approaches that co-locate tasks that communicate shared data with each other on the same core [14, 25, 52]. By co-locating communicating tasks on the same core, tasks assigned to a core better utilize the core's memory hierarchy compared to cache bypassing approaches. However, these approaches also prevent multiple copies of communicated data to be stored in different cores' private caches. On the other hand, our predictable hardware cache coherence mechanisms do not impose any data caching or application execution constraints, which results in unconstrained utilization of the memory hierarchy. For this reason, predictable hardware cache coherence mechanisms significantly outperform state-of-the-art approaches as we show in the following chapters.

Data duplication of communicated data is another common approach typically adopted by industry to eliminate timing interference due to data communication [43, 60]. The key mechanism behind these approaches is that tasks running on different cores create dedicated copies of shared data, perform operations on them, and then write-back the modified data contents. This model of communication ensures that tasks work on the same version of data throughout their execution, and their temporal behavior is not affected by other simultaneous tasks working on other dedicated copies of the same shared data. However, there are two drawbacks of this approach: (1) data duplication increases memory footprint of applications and (2) tasks may not operate on *fresh* data. The first drawback may result in higher memory storage requirements, which can go against the size, weight and power constraints imposed by embedded environments. The second drawback can degrade applications' quality-of-service (QoS) by forcing applications to operate on stale data; for example, tasks that operate on rapidly changing sensor values. On the other hand, predictable hardware cache coherence mechanisms do not duplicate shared data, and enforce rules such that communicating tasks running on different cores operate on the most up-to-date data.

2.3.3 Hardware cache coherence mechanisms

Predictable hardware cache coherence

To the best of our knowledge, the research contributions discussed in Chapter 3 was the first work to make a case for predictable hardware cache coherence for real-time multi-core platforms. This has prompted several research works that analyzed timing behavior of existing hardware cache coherence mechanisms in COTS multi-core platforms [114, 133, 135] and designed novel predictable hardware cache coherence mechanisms that offer different predictability and performance trade-offs [13, 65, 67, 139]. We describe these related works in detail in the following chapters.

Conventional hardware cache coherence

Hardware cache coherence is a *primary* feature in COTS multi-core platforms [97]. Given its importance in facilitating coherent data communication between cores, there is a rich body of research spanning different optimization targets such as design complexity [26, 81, 122, 151], scalability [3, 70, 71, 110, 142, 162, 163], high-performance [36, 37, 100, 125, 164, 168], low power [21, 93, 98, 101, 167], and security [160] to name a few. Furthermore, there is active research in extending hardware cache coherence to facilitate coherent data communication between *multiple processors* such as a multi-core processor and accelerators such as GPUs or FPGAs [6, 113, 169]. While these optimization targets are important for multi-core real-time systems, especially power and security, they do not optimize for timing predictability. The research contributions in this thesis propose new hardware cache coherence mechanisms that are optimized to improve timing predictability while retaining the performance benefits of conventional hardware cache coherence mechanisms. These prior research works on improving design complexity, high-performance, power, and security are *orthogonal* to our research contributions, and open up several avenues of future works as described in Chapter 8.

Chapter 3

Designing Predictable Cache Coherence Mechanisms for Hard Real-Time Systems

The *first research contribution* of this thesis shows *how* to design predictable shared data communication mechanisms using hardware cache coherence. As described in Section 2.2, hardware cache coherence enables multiple cores to simultaneously cache shared data in their private cache. However, COTS multi-core manufacturers disclose few details about the underlying hardware cache coherence mechanisms, which make it difficult to ascertain their timing predictability. To this end, we assume a *generic* hardware cache coherence mechanism that enables multiple cores to simultaneously cache shared data in their private cache, and examine all possible scenarios that result in timing unpredictability. Based on these timing unpredictable scenarios, we present a *design template* in the form of design guidelines for designing predictable hardware cache coherence mechanisms. Using this design template, we describe and evaluate the design of three predictable hardware cache coherence mechanisms that are amenable to timing analysis.

3.1 Introduction

In hard real-time systems, correctness depends on both the functioning behavior, and on the timing of that behavior [90]. Applications running on these systems have strict requirements on meeting their execution time deadlines. Missing a deadline in a hard real-time system may cause catastrophic failures [109]. Therefore, ensuring that deadlines are always met via static timing analysis is mandatory for such systems. Timing analysis computes an upper bound for the execution time of each running application on the system by carefully accounting for hardware implementation details, and using sophisticated abstraction techniques. The worst-case execution time

(WCET) of any application has to be less than or equal to this upper bound to achieve predictability. As application demands continue to increase from the avionics [103] and automotive [127] domains, there is increasing attention in deploying multi-core platforms. This is primarily due to the benefits multi-core platforms provide in cost, and performance. However, multi-core platforms pose new challenges towards guaranteeing temporal requirements of running applications. Among these challenges, achieving predictable shared data accesses in real-time applications has gained recent attention from the research community [25, 60, 61, 86, 153]. Recent work has showed clear evidence of data sharing between real-time tasks in practical real-time domains deployed on multi-core platforms [60], thereby making this an important challenge and the main focus of this work.

One mechanism for managing shared data accesses that is standard in existing multi-core platforms is *cache coherence* [97, 138]. Cache coherence mechanism provides all cores in the multi-core platform access to coherent data that may be cached in their private caches [138]. Cache coherence is realized by implementing a protocol that specifies a core's activity (read or write) on cached shared data based on the activity of other cores on the same shared data. While cache coherence can be implemented in software or hardware, modern multi-core platforms implement the cache coherence protocol in hardware [97]. This is so that software programmers do not have to explicitly manage coherence of shared data in the application. A recent work studied the effect of cache coherence on execution time using different Intel and AMD processors and coherence protocols [52]. The study compared execution times between executing an application sequentially and in parallel. It concluded that the interference from cache coherence can severely reduce benefits gained from parallelism. In fact, it can make parallel execution 3.87$\times$ slower than sequential execution [52]. For real-time applications that share data, this emphasizes the importance of considering cache coherence effects when deriving WCET bounds. However, as observed by a recent survey [51], there is no existing technique to account for the effects of cache coherence in static timing analysis in real-time systems.

Current techniques, which do not use cache coherence, enable coherent data sharing by enforcing restrictions on shared data accesses. These techniques include (1) disabling caching of shared data [61, 86], (2) mapping tasks that share data to execute on the same core [23, 25, 50], and (3) marking shared data accesses as critical sections such that they are accessed by a single core at any time instance [115]. These techniques enable predictable and coherent data sharing at the expense of (1) severely degrading average-case performance, (2) imposing task scheduling restrictions, and (3) application and real-time operating system (RTOS) modifications. On the other hand, a predictable hardware cache coherence mechanism can address these three limitations of current techniques as it (1) allows shared data to reside in the private caches of multiple cores simultaneously, (2) does not impose task scheduling restrictions, and (3) does not require application modifications.

This chapter describes how to design hardware cache coherence mechanisms for managing predictable shared data accesses. A key contribution of this work is to show that a simple combination of a conventional hardware cache coherence protocol and a shared bus that deploys a predictable shared bus arbitration policy is *insufficient* to guarantee predictable shared data accesses under cache coherence. We address the problem of maintaining cache coherence in multi-core real-time systems by analyzing and modifying conventional hardware cache coherence protocols. The resulting cache coherence protocols allow for predictable and coherent data sharing in a manner amenable for timing analysis [74]. This chapter analyzes the conventional MSI and MESI snooping bus-based cache coherence protocols, and describes the design of predictable variants of these protocols – predictable MSI (PMSI) and predictable MESI (PMESI) protocol respectively. The timing analysis computes the worst-case latency (WCL) of a memory request under these protocols. The timing analysis shows that although PMESI has additional performance benefits over PMSI, the WCL of a memory request under PMSI and PMESI are the same. We also identify opportunities to improve the average-case performance of PMESI through additional hardware modifications, which results in a new protocol Opt-PMESI.

3.2 Main contributions

In summary, we make the following contributions in this chapter.

1. We identify scenarios in conventional cache coherence protocols that can lead to scenarios where a memory request has unbounded memory latency (Section 3.5). The identified scenarios are general and independent of the implementation details of the deployed cache coherence protocol. Based on the scenarios, we propose a set of design invariants to address the unbounded memory latency. Implementing these design invariants results in a predictable cache coherence protocol where a memory request has bounded memory latency.

2. We analyze the conventional MSI and MESI coherence protocol, and highlight the unpredictable behaviors in these protocol. We propose extensions to the MSI and MESI coherence protocols to guarantee predictability resulting in the predicable MSI (PMSI) and predictable MESI (PMESI) protocols. The PMSI and PMESI protocols satisfy the design invariants for predictability (Section 3.5) through protocol changes and architectural extensions (Section 3.6). We also design the Opt-PMESI protocol that improves average-case performance over PMESI protocol through hardware optimizations.

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

4. We evaluate the proposed coherence protocol using the gem5 simulator [17] (Section 3.8). Performance evaluation using synthetic and SPLASH-2 workloads shows that PMSI, PMESI, and Opt-PMESI achieve up to $4\times$ speedup over competitive predictable approaches for a quad-core system while guaranteeing predictability. Furthermore, Opt-PMESI improves performance over PMSI and PMESI by up to 12%.

3.3 Related Work

Prior research efforts investigated the access latency overhead resulting from shared buses [109], caches [131, 141, 153], and dynamic random access memories (DRAMs) [64, 118, 159]. For shared caches, most of these efforts primarily focused on preventing a task's data accesses from affecting another task's data accesses. They used data isolation between tasks by utilizing strict cache partitioning [153] or locking mechanisms [141]. Authors in [131] promoted splitting the data cache into multiple data regions that simplified the analysis. However, they indicated that cache coherence is still an issue that has to be addressed. Similarly, several proposals for shared main memories deployed data isolation that assigned a private memory bank per core [118, 159]. However, we find that data isolation suffers from three limitations. The first limitation is that it disallows sharing of data between tasks; thus, disabling any communication across applications or threads of parallel tasks running on different cores. The second limitation is that it may result in poor memory or cache utilization. For instance, a task may keep evicting its cache lines if it reaches the maximum of its partition size, while other partitions may remain underutilized. The third limitation is that it does not scale with increasing number of cores. For example, the number of cores in the system has to be less than or equal to the number of DRAM banks to be able to achieve isolation at DRAM.

Recent works [64, 87] recognized these limitations, and offered solutions for sharing data. Authors in [64] shared the whole memory space between tasks for main memory, and [87] suggested a compromise that divided the memory space into private and shared segments for caches. Nonetheless, these approaches focused on the impact of sharing memory on timing analysis, and they did not address the problem of data correctness resulting from sharing memory. Authors of [16] studied the overhead effects of co-running applications on the timing behavior in the avionics domain, where cache coherence was one of the overhead sources. A recent survey [51] observed that there is no existing technique to include the effects of data coherence on timing analysis for multi-core real-time systems.

However, there exist approaches that attempt to eliminate unpredictable scenarios that arise from data sharing. Authors in [23] proposed data sharing-aware scheduling policies that avoided running tasks with shared data simultaneously. A similar approach proposed by [50, 52] re-designed the real-time operating system to include cache partitioning, task scheduling, and feedback from the performance counters to account for cache coherence in task scheduling decisions. Such approaches rely on hardware counters that feed the schedule with information about memory requests. They also require modifications to existing task scheduling techniques. For example, the solution in [23] is not adequate for partitioned scheduling mechanisms. A different solution introduced in [115] applied source-code modifications to mark instructions with shared data as critical sections. These critical sections were protected by locking mechanisms such that they were accessed only by a single core at any time instance. This solution suffers from two limitations. The first limitation is that the software is responsible for maintaining cache coherence to guarantee shared data correctness. As a result, additional changes to the software are necessary in order to explicitly manage cache coherence. The second limitation is that only one core can access a cache line of shared data at a time. Other cores requesting this data must wait until a core completes all operations on the shared data. In the worst case, this is equivalent to sequential execution. On the other hand, our proposed cache coherence protocols (PMSI and PMESI) allow tasks to simultaneously access shared data, which considerably improves performance. In addition, PMSI and PMESI do not pose any requirements on task scheduling techniques, and they do not require software modifications.

3.4 System Model

We consider a multi-core system with N cores, $\{c_0,c_1,...,c_{N-1}\}$. Each core has a private cache, and all cores have access to a shared memory. This shared memory can be an on-chip last-level cache (LLC), an off-chip DRAM, or both. Tasks running on cores share data. These tasks can belong to a parallel application that is distributed across cores, or different applications that communicate between each other. Cores can share the whole shared memory space similar to [64] or share part of the memory space similar to [87]. We do not impose any restrictions on how the interference on the shared memory is resolved, whether it is the LLC or the DRAM. Furthermore, we do not require any special demands from the task scheduling mechanism. This allows one to integrate the proposed solution to current task scheduling techniques, and to various mechanisms that control accesses to shared memories in multi-core real-time systems. Cores share a common snooping bus that connects private caches of cores to the shared memory. This shared bus allows for cores to broadcast their memory requests to other cores and the shared memory, and data transfers between the shared memory and cores. The shared bus also transfers

coherence messages deployed by the coherence protocol to ensure data correctness. Cores *snoop* the bus to observe memory activity of other cores. The system deploys a predictable arbitration on the shared bus. Note that data transfers between private caches are only via the shared memory (no cache-to-cache transfers). The proposed solution is independent of the core architecture, and the predictable arbitration mechanism on the bus. However, the analysis and experiments we present in this work consider a system with in-order cores, and a time-division-multiplexing (TDM) bus as the base arbitration scheme. A TDM slot width allows for one data transfer between shared memory and the private cache including the overhead of necessary coherence messages.

3.5 Design Invariants for Predictable Cache Coherence

A cache coherence protocol ensures correctness of shared data across all cores in a multi-core platform. As we show in this section, simply adopting a predictable arbiter in this case does not necessarily mean that tasks will have predictable latencies upon accessing the shared memory. This is because the latency suffered by one core accessing a shared line is dependent on the coherence state of that line in the private caches of other cores. Two major contributions of this paper are (1) to identify these unpredictable scenarios, and (2) to propose invariants to address them. In this section, we describe these unpredictable scenarios, and propose design invariants to address these scenarios. Exact sources of unpredictability in current multi-core platforms are dependent on the cache coherence protocol and micro-architecture details of the cache controllers, which are proprietary and are not publicly available. The proposed invariants are general design guidelines, which are independent of the adopted cache coherence protocol implementation and the underlying platform architecture. For the predictable cache coherence protocols described in Section 3.6, we realize these invariants using a combination of cache coherence protocol changes and hardware structures.

An arbiter manages accesses to the shared bus such that at any time instance it exclusively grants bus access to a single core. A predictable arbiter guarantees that each requesting core is granted the bus eventually in a defined upper-bound amount of time. Upon implementing a coherence protocol, a core initiates memory requests by exchanging coherence messages with other cores and the shared memory. Therefore, before investigating the potential sources of unpredictability, we extend the predictable bus arbiter with Invariant 1 such that it manages *both* data transfers and coherence messages.

Invariant 1. ***A predictable bus arbiter must manage coherence messages and data on the bus such that each core broadcasts a coherence request or communicates data on the bus if and only if it is granted an access slot to the bus.***

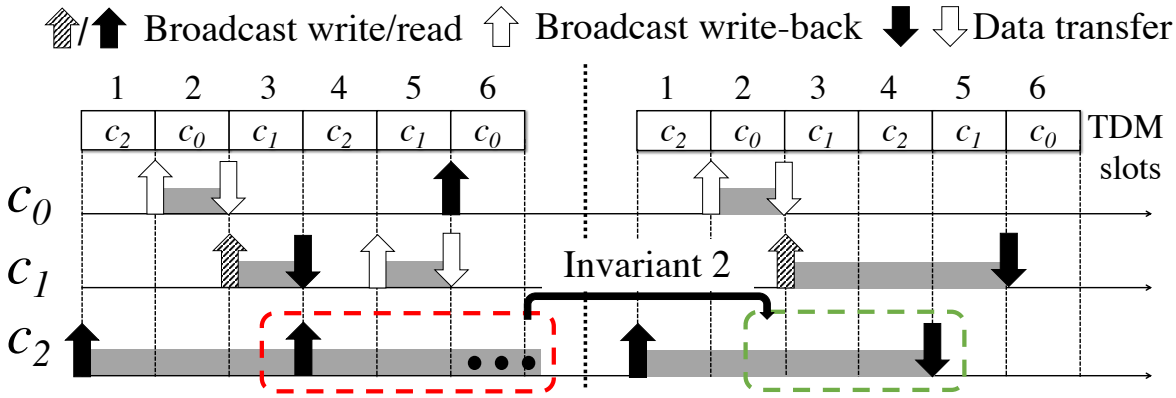


Figure 3.1: Initially c_0 modified A. c_2 is core under analysis.

Investigating the implications of a conventional coherence protocol on the WCET, we find that there are **five** major sources that can lead to unpredictable behavior. We group these sources into two categories: *inter-core interference* and *intra-core interference*. Figures 3.1–3.4 illustrate example scenarios for these sources. The example scenarios consider a system with three cores, c_0, c_1, and c_2, and deploys a TDM arbitration across cores. If the request type is not specified whether it is a read or write, that means the scenario is agnostic to it. Each of Figures 3.1–3.4 separately defines the initial system state and the core under analysis for the corresponding scenario. We denote TDM slot i as ❶.

3.5.1 Inter-core Coherence Interference

Inter-core interference arises due to memory activity across *different* cores. We enumerate four unpredictable scenarios that arise due to memory activity across different cores. These scenarios differ based on (1) the memory activity (reads/writes), and (2) the cache lines accessed by the cores.

Interference on same line

The first source of unpredictability arises from multiple cores reading the same modified cache line, say A. If a core requests to modify A, it has to wait for the owner to write-back A to the shared memory. In Figure 3.1, initially, c_0 has a modified version of A in its private cache. The core under analysis is c_2. At ❶, c_2 broadcasts a read request to A. Since c_0 has the modified version of A, it has to write-back the updated A to the shared memory first. However, this is c_2's slot; thus, c_0 has to wait for its allocated slot to perform the write-back. Hence, at ❷, c_0 writes

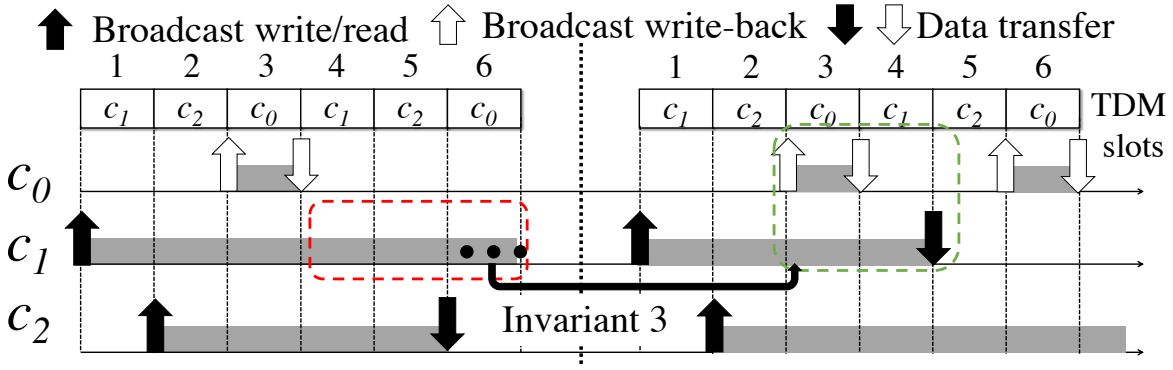


Figure 3.2: Initially c_0 modified A and B. c_1 is core under analysis.

back A to the shared memory in its slot. At ❸, c_1 broadcasts a write request to A. Since the shared memory has the updated version of A, c_1 is able to obtain A and modify it. As a result, c_2 re-broadcasts a read request to A at ❹. This time c_2 has to wait for c_1 to write-back A. From c_2's perspective, the events at ❹ are a repetition of the events at ❶; c_2 re-broadcasts its request to A and waits for another core to write it back. Thus, this situation is repeatable and can result in unbounded memory latency. Although c_2 is granted access to the bus, it is unable to obtain the requested data due to the coherence interference.

Invariant 2. ***The shared memory services requests to the same line in the order of their arrival to the shared memory.***

Proposed solution. Invariant 2 requires memory to service requests to the same cache line in their *arrival order*; thus, it guarantees that a line being requested by a core will not be invalidated before the core accesses it. In the above example, the memory serviced requests based on the arbitration schedule order, which is different from the arrival order resulting in unbounded memory latency. Imposing Invariant 2 in Figure 3.1, c_2's request to A arrives to the shared memory before c_1's request; therefore, c_1 has to wait for c_2 to execute its operation before it gains an access to A. Note that in conventional snooping bus-based coherence protocols, this invariant is realized by ensuring that the shared memory responds to memory requests from cores based on their broadcasted order [138].

Interference on different lines

The second source of interference arises when multiple cores request different cache lines that are modified by the same core (owner). As a result, the owner has to write-back the modified lines requested by the other cores to the shared memory. For instance in Figure 3.2, c_0 has

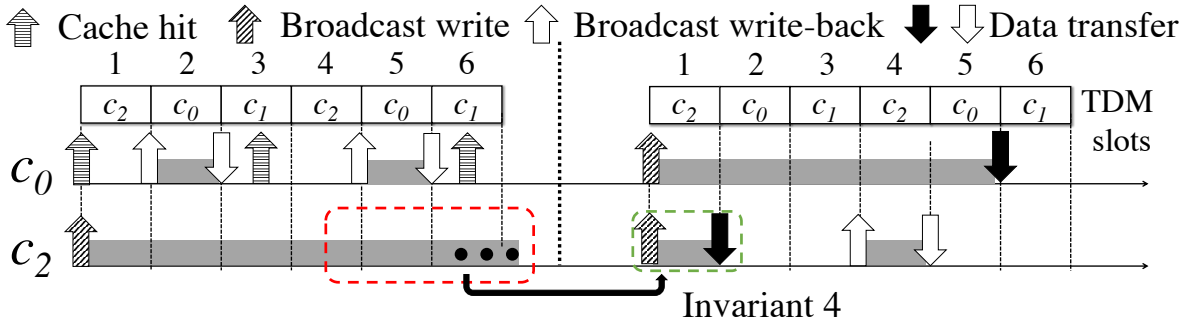


Figure 3.3: Initially c_0 reads A. c_2 is core under analysis.

modified versions of lines A and B. The core under analysis is c_1. c_1 broadcasts a request to A in ❶, and c_2 broadcasts a request to B in ❷. Accordingly, c_0 has to write-back both A and B to the shared memory. Since c_0 can schedule one memory transfer in a slot, it can write-back only one line to the shared memory. If no predictable mechanism manages the write-backs, c_0 can pick any pending one. At ❸, c_0 writes back B. Therefore, at ❹, c_1 is stalled on A. This situation can repeat indefinitely. While c_1 is waiting for A, c_2 can ask for another line, which is also modified by c_0 and the same situation can repeat.

Invariant 3. ***A core responds to coherence requests in the order of their arrival to that core.***

Proposed solution. Invariant 3 imposes an order in servicing coherence messages from other cores (write-backs, for example). The right side of Figure 3.2 deploys Invariant 3. Since the request to A arrives before that to B, c_0 has to write-back A first (in ❸) then B (in ❻); thus, a predictable behavior is guaranteed.

Writes to non-modified lines

The third source is due to write hits in the private cache to non-modified lines. Recall that the predictable bus arbiter only controls accesses to the shared bus. As a result, a request that results in a hit in the private cache can proceed without waiting for the corresponding core slot. However, write requests to unmodified lines that hit in the private cache can result in the following two unpredictable scenarios described in Figures 3.3 and 3.4.

The first scenario arises when multiple cores update the same line and one of the cores has the line in an unmodified state in its private cache. For example, in Figure 3.3, c_0 has a version of A in its private cache that is not modified. At ❶, c_2 broadcasts a read request to A, while simultaneously c_0 has a write operation to A that results in a hit in its private cache. Consider the

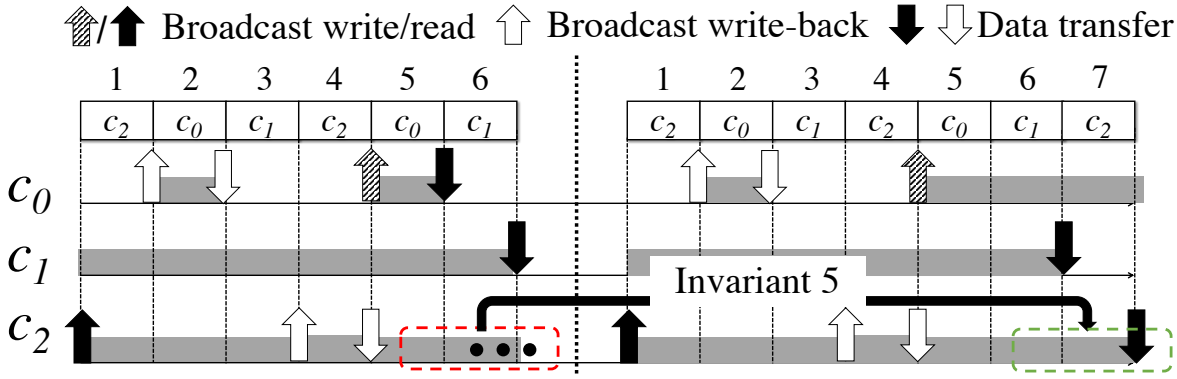


Figure 3.4: Initially c_0 modified A, c_2 modified B, and c_1 requested B. c_2 is core under analysis.

scenario where c_0's write hit on A occurs first. As a result, c_2 has to wait until c_0 writes back A. This scenario is shown in Figure 3.3. After c_0 writes back A in ❷, c_0 again has another write hit to A in ❸. Again, c_2 has to wait for c_0 to write-back A. Consequently, this situation is repeatable and can starve c_2.

Invariant 4. ***A write request from c_i that is a hit to a non-modified line in c_i's private cache has to wait for the arbiter to grant c_i an access to the bus.***

Proposed solution. Invariant 4 stalls a write request by a core, which is a hit to a non-modified line until the arbiter grants an access slot to that core. Thereby, it avoids the aforementioned unpredictable consequences. It is worth noting that Invariant 4 aligns with Invariant 1 as follows. Invariant 1 mandates that a core can initiate coherence messages into the bus only when it is granted an access to it by the arbiter. Although a write hit to a non-modified line does not need data from the shared memory, it still needs to send coherence messages on the bus. This is necessary to invalidate local copies of the same line that other cores have in their private caches. Accordingly, a write hit to a non-modified line has to wait for a granted access by the arbiter. On maintaining Invariant 4 in Figure 3.3, the following behavior is guaranteed. Since ❶ belongs to c_2, and c_0's request is a write hit to A, which is not modified, c_0 must wait for its slot to that request. c_2 broadcasts its write request to A in ❶, and c_0 invalidates its own local copy of A. Since no core has a modified version of A, c_2 obtains A from the shared memory and performs the write operation.

Invariant 4 resolves the race situation between a request generated by a core in its designated slot and write hits from other cores. However, a second unpredictable scenario is possible that Invariant 4 does not manage. We describe this scenario using Figure 3.4. Initially, c_0 has a modified version of A, c_2 has a modified version of B, and c_1 has requested B. At ❶, c_2 broadcasts a read request to A; thus, c_0 updates the shared memory with the modified value of A at ❷. Since

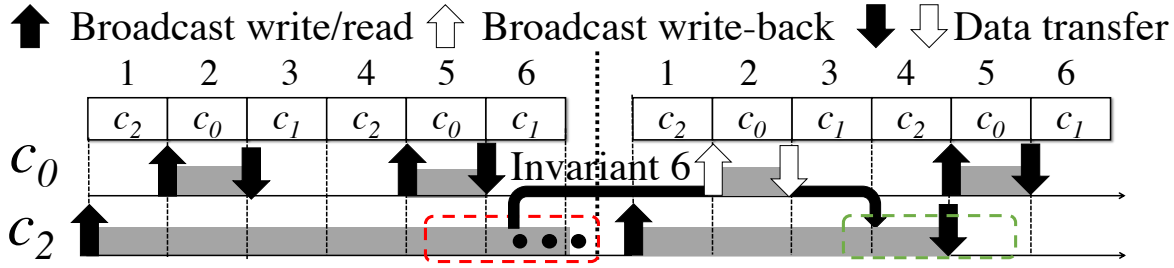


Figure 3.5: Initially, c_0 has modified A. c_2 is under analysis.

c_2's request is a read, c_0 does not invalidate its local version of A. At ❹, c_2 has two pending actions: fetching A from memory, and writing back B to the memory in response to c_1's request. Assume that c_2 chooses to write-back B. Therefore, its request to A waits for the next slot. At ❺, c_0 has a write hit to A. Consequently, since this is c_0's slot, it conforms with Invariant 4; thereby, it modifies A. At ❻, c_2 has to re-broadcast its request to A and wait for c_0 to write-back A to memory again. From c_2's perspective, this situation is similar to the situation at ❶. Similarly, in subsequent periods, after c_0 writes back A, it can have a write hit to A before c_2 receives it from the memory. Clearly, this situation is repeatable indefinitely, and creates unbounded memory latency for c_2.

Invariant 5. ***A write request from c_i that is a hit to a non-modified line, say A, in c_i's private cache has to wait until all waiting cores that previously requested A get an access to A.***

Proposed solution. Invariant 5 stalls a write request to a non-modified line until all pending requests from previous slots are completed. Thereby, it avoids the above unpredictable scenario. Maintaining Invariant 5 in the right side of Figure 3.4, the following behavior is guaranteed. During c_0's slot, it has a hit to A. Since A is non-modified by c_0 and is previously requested by c_2, the write hit cannot be processed. Accordingly, c_2 obtains A from the shared memory in its next slot and performs its operation. c_0's request to A is broadcasted afterwards in the corresponding slot.

3.5.2 Intra-core Coherence Interference

Intra-core coherence interference arises due to multiple memory activity from the *same* core such as a core's own pending request and its response to a request from another core. This response is for example, a write-back to a line that this core has in a modified state. In Figure 3.5, c_0 has a modified version of A. At ❶, c_2 broadcasts a request to A; thus, c_0 marks A for write-back in its next slot. However, at ❷, c_0 has a pending request to B that is broadcasted in ❷. Thus,

the write-back of A waits for c_0's next slot. Similarly, at ❹, c_0 has another pending request to another line, C. Accordingly, the write-back of A by c_0 can indefinitely stall, which results in unbounded latency of c_2's request.

Invariant 6. ***Each core has to deploy a predictable arbitration between its own generated requests and its responses to requests from other cores.***

Proposed solution. Invariant 6 states that any predictable arbitration mechanism between coherence requests of a core and responses from the same core is sufficient to address the intra-core interference. Deciding the adequate arbitration depends on the application. Deploying Invariant 6 in Figure 3.5, the predictable arbitration mechanism will eventually allocate one slot to c_0's write-back operation of A, which bounds the memory latency of c_1's request.

3.6 Predictable Cache Coherence Protocols

We show the effectiveness of the proposed invariants by applying them to the conventional MSI and MESI protocols. This results in predictable MSI (PMSI) and predictable MESI (PMESI) protocols for multi-core real-time systems. To ensure that the invariants described in Section 3.5 are held, we propose architectural modifications and additional coherence states. The architectural modifications apply to both PMSI and PMESI, and the additional coherence states are protocol specific.

Invariants require either architectural modifications or a combination of both architectural modifications and additional coherence states. For example, Invariants 1 and 2 require only architectural modifications and no changes to the coherence protocols. On the other hand, Invariants 3–6 require modifications to both the architecture and the coherence protocol. This is because Invariants 3 and 6 regulate the write-back operation of cache lines. Since a core has to wait for a designated write-back slot to write-back a cache line A, it has to maintain A in a t-state to indicate that A is waiting for write-back. Similarly, Invariants 4 and 5 regulate the write hit operation to non-modified lines. A core has to wait for a designated slot to perform the write hit operation to a cache line, say B. Accordingly, it has to maintain B in a t-state indicating that it has a pending write to B.

In the following sections, we describe the architectural modifications that are required for both PMSI and PMESI coherence protocols (Section 3.6.1), and then describe the PMSI and PMESI protocol modifications (Section 3.6.2).

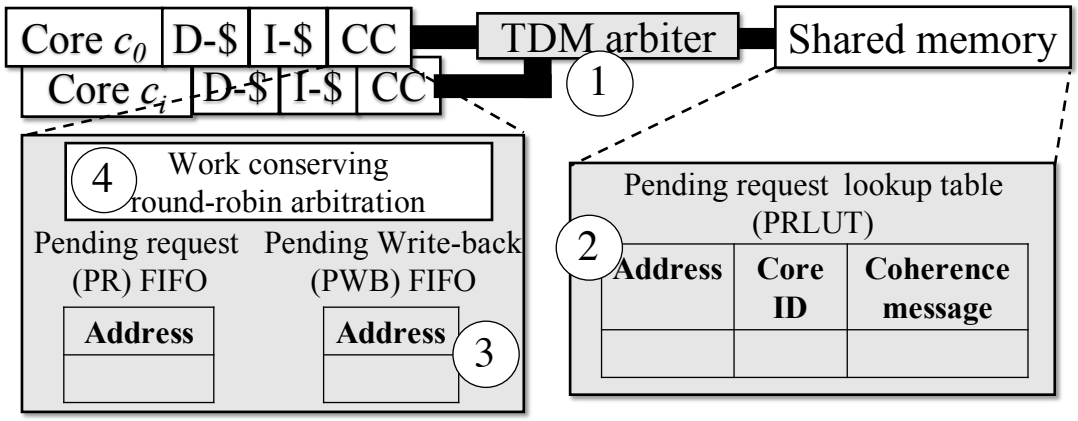


Figure 3.6: Architectural changes necessary for PMSI and PMESI.

3.6.1 Architectural Modifications

Figure 3.6 depicts a multi-core system with a private cache for each core and a shared memory connected to all cores via a shared bus. A TDM bus arbiter manages accesses to the shared memory. The proposed architecture changes are highlighted in grey.

The TDM arbiter ① manages the coherence requests such that each core can issue a coherence request message only when it is granted an access to the bus. This satisfies Invariant 1. The shared memory uses a *first-in-first-out* (FIFO) arbitration between requests to the same cache line. We implement this arbitration using a look-up table (LUT) ② to queue pending requests (PR), denoted as PR LUT in Figure 3.6. Each entry consists of the address of the requested line, the identification of the requesting core, and the coherence message. The PR LUT queues requests by the order of their arrival. When the memory has the updated data of a cache line, it services the oldest pending request for that line. Each core buffers the pending write-back responses in a FIFO queue, which Figure 3.6 denotes as the pending write-back (PWB) FIFO ③. This modification cooperates with the proposed t-states to satisfy Invariant 3. Each core deploys a work-conserving TDM arbitration between the PR and PWB FIFOs ④. This arbitration along with the proposed t-states comply with Invariant 6.

These architectural changes, along with the coherence protocol changes, also satisfy Invariants 4 and 5 as follows. If a core c_i has a write hit to a non-modified line A, it has to broadcast an Upg() coherence message on the bus. With ①, the arbiter does not allow this Upg() message on the bus unless it is the TDM slot of the initiating core. In consequence, the write hit to A is postponed to c_i's next slot, which implements Invariant 4. Assume that during c_i's next slot, there were one or more pending requests to A from other cores that arrived before c_i's request. According to Invariant 5, c_i's write hit to A has to wait until these pending requests are serviced.

Core count	PRB (bits)	PWB (bits)	PRLUT (bits)	Total (bytes)
2	128	128	134	81
4	256	256	272	290
8	512	512	552	1093
16	1024	1024	1120	4236

Table 3.1: Hardware overheads with core count.

Recall that PR LUT ② queues pending requests. If the write hit is to one of these lines, the arbiter does not select the store hit to execute during this slot. Accordingly, Invariant 5 is fulfilled.

Hardware overhead. For a N-core system, the PR, PWB, and PRLUT structures have N entries each as each core can only have one pending request at any time instance. For an address width of 64-bits, the hardware overheads of the per-core PR, per-core PWB, and PRLUT buffers are $64 \times N$-bits, $64 \times N$-bits, and $(64 + log_2 N + 2) \times N$-bits respectively; the core ID and coherence message fields in the PRLUT are $log_2 N$-bits and 2-bits wide respectively. Hence, for a 4-core, 8-core, and 16-core system, the total hardware overheads are 290-bytes, 1093-bytes and 4236-bytes respectively. Table 3.1 describes the per structure hardware overhead for different core counts.

	Core events			Bus events					
	Read	Write	Replacement	OwnData	OwnUpg	OwnPutM	OtherGetS	OtherGetM/OtherUpg	OtherPutM
I	Issue GetS/IS_D	Issue GetM/IM_D					—	—	—
S	Hit, Complete read	Issue Upg/SM_D	I/-				—	-/I	
M	Hit, Complete read	Hit, Complete write	Issue PutM/MI_A			Issue PutM/MS_A	Issue PutM/MI_A		
IS_D				If E-Read, Complete read/E, else Complete read/S			—	-/IS_DI	—
IM_D				Complete write/M			-/IM_DS	-/IM_DI	—
SM_A			Stall		Complete write/M		—	Reissue write/I	
MI_A	Hit, Complete read	Hit, Complete write	—			Write-back to memory/I	—	—	
MS_A	Hit, Complete read	Hit, Complete write	-/MI_A			Write-back data to memory/S	—	-/MI_A	
IM_DI				Complete write, issue PutM/MI_A			—	—	—
IS_DI				Complete read/I			—	—	—
IM_DS				Complete write, issue PutM/MS_A			—	-/IM_DI	—
E	Hit, Complete read	Hit, Complete write/M	(A) Issue PutM/EI_A (B) Send NoData to memory/I				(A) Issue PutM/ES_A (B) Send NoData to memory/S	(A) Issue PutM/EI_A (B) Send NoData to memory/I	
EI_A	Hit, Complete read	Hit, Complete write/MI_A	—			Send data to memory/I	—	—	
ES_A	Hit, Complete read	Hit, Complete write/MS_A	-/EI_A			Send data to memory/S	—	-/EI_A	

Table 3.2: Private memory states for PMSI, PMESI, and Opt-PMESI. *issue msg/state* means the core issues the message *msg* and move to state *state*. A core issues a *read/write* request. Once the cache line is available, the core *reads/writes* it. A core needs to issue a *replacement* to write back a dirty block before eviction. Changes to conventional MSI and MESI are in bold red. Differing transitions between PMESI and Opt-PMESI are marked as (A) and (B) respectively.

State	Core events			
	GetS	GetM	PutM	Data from core
IoS	Send data to requesting core/IoS	Send data to requesting core/M	—	
M	Clear owner/IoS_D	Update owner to requesting core/M	Clear owner/IoS_D	
IoS_D	Stall	Stall	Stall	Write memory/IoS

Table 3.3: Shared memory states for PMSI protocol.

State	Core events			
	GetS	GetM	PutM	Data from core
I	Send data to requesting core/M	Send data to requesting core/M	—	
S	Send data to requesting core/S	Send data to requesting core/M	—	
M	Clear owner/S_D	Update owner to requesting core/M	Clear owner/IoS_D	
S_D	Stall	Stall	Stall	Write memory/S

Table 3.4: Shared memory states for PMESI protocol.

State	Core events				
	GetS	GetM	PutM	Data from core	NoData from core
I	Send data to requesting core/M	Send data to requesting core/M	—		
S	Send data to requesting core/S	Send data to requesting core/M	—		
M	Clear owner/S_D	Update owner to requesting core/M	Clear owner/IoS_D		
S_D	Stall	Stall	Stall	Write memory/S	-/S

Table 3.5: Shared memory states for Opt-PMESI protocol.

3.6.2 Cache coherence protocol state machine modifications

We discuss the protocol modifications to the MSI and MESI protocols that work in tandem with the hardware structures described earlier. The protocol modifications result in new protocols: PMSI and PMESI protocols respectively. We also describe an optimized variant of PMESI, Opt-PMESI, which adds hardware and protocol extensions to improve the average-case performance of PMESI. We first describe the t-states that are removed and unmodified across all the protocols (Sections 3.6.2 and 3.6.2), and then introduce new t-states introduced for each protocol in Sections 3.6.2-3.6.2. Table 3.2 shows the private caches' coherence states for a cache line and the transitions between these states for the PMSI, PMESI, and Opt-PMESI protocols, and Tables 3.3-3.5 shows the shared memory coherence states and transitions for the PMSI, PMESI, and Opt-PMESI protocols. Table 3.6 describes the new t-states added to the PMSI and PMESI protocols. We do not make changes to the coherence states for the shared memory, and hence it is not shown. Shaded cells represent transitions that are not possible under correct operation. Cells marked with "-" represent situations where no transition occurs, and the coherence state remains unchanged.

Removed t-states

For a real-time system, t-states that indicate unavailability of cache line in the private caches and waiting for coherence messages to appear on the bus are not needed. Examples of such

Initial s-state	Transient state	Final s-state	Description
M	MI_A	I	c_i has a line A in M state. Another core requested A to modify. MI_A is necessary to reflect that c_i has to write-back A in its next write-back slot.
E	EI_A	I	c_i has a line A in E state. Another core requested A to modify. EI_A is necessary to reflect that c_i has to write-back A in its next write-back slot.
M	MS_A	S	Similar to MI_A except that the other core requested a read to A.
E	ES_A	S	Similar to EI_A except that the other core requested a read to A.

Table 3.6: Description of the proposed t-states in PMSI and PMESI to achieve a predictable behavior.

t-states include IM_AD, IS_AD, IM_A, and IS_A (Section 2.2, Chapter 2). On deploying a predictable bus arbitration, once a core is granted access to the bus, no other core can issue a coherence message during that slot. This is assured by Invariant 1. Accordingly, during a core's slot, its coherence messages are not disrupted by messages from other cores. By removing these t-states, PMSI, PMESI, and Opt-PMESI has fewer states and transitions compared to their respective conventional protocols [138]. For example, assume that c_i issues a read request to a line A that is invalid in its private cache. During c_i's slot, it broadcasts its OwnGetS() to the bus. Since c_i is the only core broadcasting coherence messages to the bus, it cannot receive its data before observing its OwnGetS() on the bus. Therefore, c_i changes A's state from I to IS_D without the need to move to IS_A. By removing these t-states, PMSI, PMESI, and Opt-PMESI has fewer states and transitions compared to their respective conventional protocols [138].

Unmodified t-states

Transient states that denote the waiting for data response are retained. Examples of such t-states are IS_D and IM_D that denote a core's read or write request is waiting for data respectively. This is because if c_i issues a request to a cache line that is modified by another core c_j, c_i must wait until c_j writes back that cache line to the shared memory. Accordingly, c_i has to move to a t-state indicating that it is waiting for a data response from the memory. In addition, there are

three other unmodified t-states such as IS_DI, IM_DI, and IM_DS. These states indicate that the core has to take an action after receiving the data and perform the operation. For example, the t-state IS_DI indicates that a core waiting on data for a broadcasted read operation observed a remote write operation. The core on receiving the data completes the read operation, invalidates its copy, and moves to the I state.

Predictable MSI cache coherence protocol (PMSI)

For PMSI protocol, we propose two additional t-states that are necessary to guarantee that invariants are upheld. t-states MI_A and MS_A manage the write-back operation for lines in the M state. These t-states convey that: (1) a line has a pending write-back response, and (2) the final state the line transitions to after the core completes the write-back. A core that has a cache line in M moves to MI_A or MS_A on observing a remote write or read request to the same cache line respectively. In the core's write-back allocated slot, the core completes the write-back and transitions to the I or S state respectively.

Figure 3.7 illustrates an example of the t-statesMI_A and MS_A. In Figure 3.7, c_0 has a write hit to an unmodified copy of A, and moves from state S to state M in ❷. c_1 broadcasts a read request to A in ❸ and moves from I to IS_D, which denotes waiting for data. c_0 observes the remote read request, and marks A for write-back, and moves from M state to MS_A state. This state indicates that c_0, in the next designated slot, will write-back A to the memory and change its local copy of A to S state. Before c_0 writes back A to shared memory, it observes c_2's modify request to A in ❹. As a consequence, it updates its A's state to MI_A, which indicates that c_0 has to invalidate A once it performs the write-back operation ❻. t-state SM_A is necessary to handle write hits to non-modified lines predictably. For example, in Figure 3.7, during c_2's slot ❶, c_0 has a write hit to A, which it has in S state. To impose Invariant 4, c_0 has to postpone this operation to its next slot. Towards doing so, it updates its A's state to SM_A to preserve the information of the upgrade request to A. In its next slot, if no other core is pending on A (Invariant 5), c_0 broadcasts its write operation on the bus, performs the store to A, and moves its A to the s-state M ❷.

Predictable MESI cache coherence protocol (PMESI)

The PMESI protocol adds the exclusive (E) state to the PMSI protocol. Section 2.2 in Chapter 2 describes the performance benefits of the E state in the MESI cache coherence protocol. Table 3.2 shows the transition to E state. Adding the E state introduces two new t-states: EI_A and ES_A states. These t-states manage the write-back operations for lines in the E state. Similar to the MI_A and MS_A t-states in PMSI, these states convey that a line has a pending write-back,

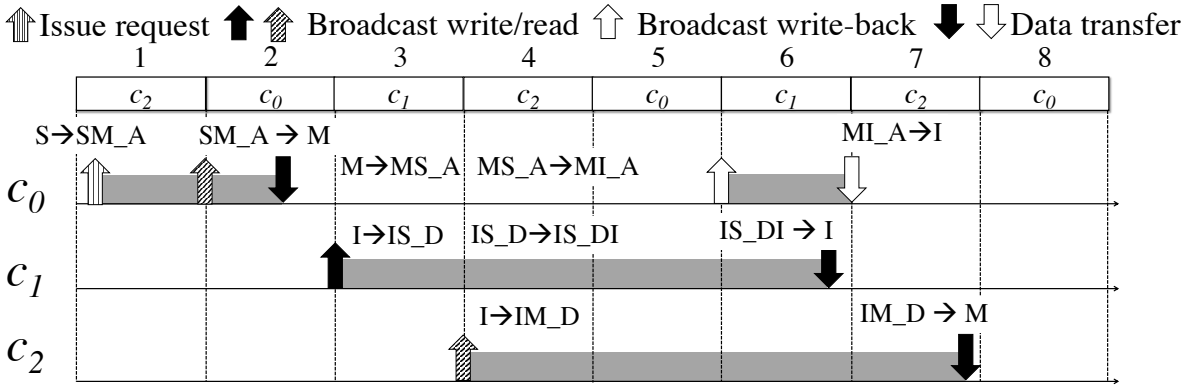
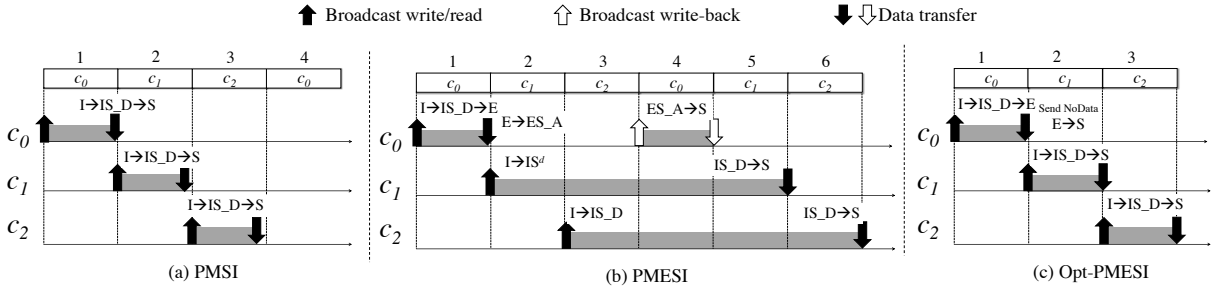


Figure 3.7: Execution with t-states. Initially, c_0 has A in **S**.



(a) PMSI (b) PMESI (c) Opt-PMESI

Figure 3.8: Execution example with PMSI and PMESI.

and the final state of the line after the write-back is completed. The rationale behind these states is that when the shared memory sends exclusive data for a requested line to a core, the coherence state of this line in the shared memory is recorded as **M**. This enables the silent writes optimization in MESI/PMESI. Hence, the shared memory cannot send data to subsequent requests to this line until the core that has the line in **E** state performs a write-back of the line. As a result, a core that has a line in **E** state, and observes remote activity on the line must mark the line for write-back.

Optimized PMESI (Opt-PMESI)

The presented PMESI protocol requires cores that have lines in states **E** or **M** to issue write-back responses to the shared memory on (1) observing remote memory activity to these lines and (2) cache line replacements. On the other hand, the PMSI protocol only performs write-back responses for lines in **M** state. As a result, lines in PMESI are subjected to more write-back responses compared to PMSI, which in turn increases request latencies for certain types of memory access patterns. This is because write-back responses and pending demand requests contend for a core's allocated slots. As a result, this can offset the performance advantage provided by silent

writes in PMESI. Figure 3.8 shows a simple scenario where cores c_0, c_1 and c_2 broadcast read requests (in that order) to a line X under PMESI and PMSI, and highlights the increased request latencies in PMESI compared to PMSI.

Figure 3.8a shows the request latencies of c_0, c_1 and c_2 to X under PMESI. Under PMESI, c_0 receives X in ❶ that is marked as exclusive as no other core has a copy of X. As a result, X in c_0's private cache is in **E** state. On observing c_1's remote read in ❷, c_0 marks X for write-back by moving from **E** state to **ES_A**. c_0 completes the write-back in ❹, and the memory moves from **M** state to **S** state as there are multiple sharers of X. c_1 receives X in ❺, and c_2 receives X in ❻ as c_2 broadcasted its request to X after c_1. Note that both c_1 and c_2 receive X in **S** state. On the other hand, under PMSI, c_0, c_1, and c_2 receive X at the end of ❶, ❷, and ❸ respectively. This execution is shown in Figure 3.8b. This is because all cores receive X in **S** state, and remote activity observed for a line in **S** state does not result in write-back responses. As a result, for this execution, PMSI offers better average-case performance than PMESI.

To address this performance limitation of PMESI, we add hardware and protocol extensions to PMESI, resulting in a new protocol that we refer to as Opt-PMESI. The key observation behind Opt-PMESI is that a line in **E** state has read-only permissions, and hence, the data contents of a line in **E** state are equal to that in the shared memory. As a result, there is no need to write-back the data contents of a line in **E** state to the shared memory. However, the shared memory tracks this line in **M** state, and hence, the core must communicate to the shared memory that it did not update this line. To facilitate this communication, we extend the shared bus to allow for an additional wire per core that is asserted by cores to communicate to the shared memory that a line in **E** state is not modified. A core asserts a signal on this wire (1) when it observes remote memory activity on a line that it has in the **E** state, and changes state immediately to either **S** or **I** based on the remote memory activity, and (2) on cache line replacements. The action of asserting a signal on this wire by a core is shown in Table 3.2 as *Send NoData to memory*. The shared memory on observing this signal assertion accordingly changes the state of the line, and responds to pending memory requests to the same line. As a result, t-states **EI_A** and **ES_A** are no longer needed. Note that this additional wire in the shared bus need not be subjected to a predictable arbitration as there can be only one core that has a line in **E** state. Hence, at most one core will assert a signal on the wire in a slot.

Figure 3.8c shows the same example with Opt-PMESI. The key difference is in ❷, where c_0 immediately moves from **E** state to **S** state on observing the remote read from c_1 to X. The shared memory on observing the asserted signal by c_0 moves from **M** to **S** state, and sends X to c_1 in ❷.

3.7 Latency Analysis

We derive the upper bound per-request latency that a core suffers when it attempts to access the shared memory. The considered system deploys one of the four predictable protocols: (1) PMSI, (2) PMESI, and (3) Opt-PMESI. Shared memory accesses from multiple cores are handled by a TDM bus arbitration scheme. We partition this latency into four components and compute the worst-case (WC) value of each of them. Definitions 1–5 formally define these latency components. We use c_i as the core under analysis, and denote a request generated by c_i as req_i.

Definition 1. ***Arbitration latency***, *L_i^{arb}, of a request req_i is measured from the time stamp of its issuance until it is granted access to the bus. L_i^{arb} is due to the arbitration schedule that allocates slots to cores.*

Definition 2. ***Access latency*** *is the time required to transfer the requested data by c_i between the shared memory and the private cache of c_i. We assume that this data transfer takes a fixed latency, L^{acc}. This latency can be considered as the WC access latency of the shared memory. Prior works such as [64, 153] can be used to determine the value of L^{acc} for LLCs and DRAMs.*

Definition 3. ***Coherence latency***, *L_i^{coh}, of a request req_i is measured from the time stamp when c_i is granted access to the bus until it starts its data transfer. L_i^{coh} is due to the deployed coherence protocol. We divide the coherence latency into two components: inter-core and intra-core coherence latency, which we denote receptively as $L_i^{interCoh}$ and $L_i^{intraCoh}$.*

Definition 4. ***Inter-core coherence latency***, *$L_i^{interCoh}$, of a request req_i is measured from the time stamp when req_i is granted access to the bus until the data is ready by the shared memory for c_i to receive in c_i's slot.*

Definition 5. *A request req_i suffers* ***intra-core coherence latency***, *$L_i^{intraCoh}$, if it has to wait until c_i issues a coherence response to an earlier request by another core. c_i is required to issue a coherence response when another core requests a line, say* B*, that c_i has in a modified state. Therefore, c_i needs to write back* B *to the shared memory.*

Lemma 1. *The WC arbitration latency, WCL_i^{arb}, of any request generated by c_i occurs when c_i has to wait for the maximum possible number of requests generated by other cores before it can issue a request on the bus. For a system deploying conventional TDM bus arbitration, WCL_i^{arb} is calculated by Equation 3.1, where N is the number of cores and S is the TDM slot width in cycles.*

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

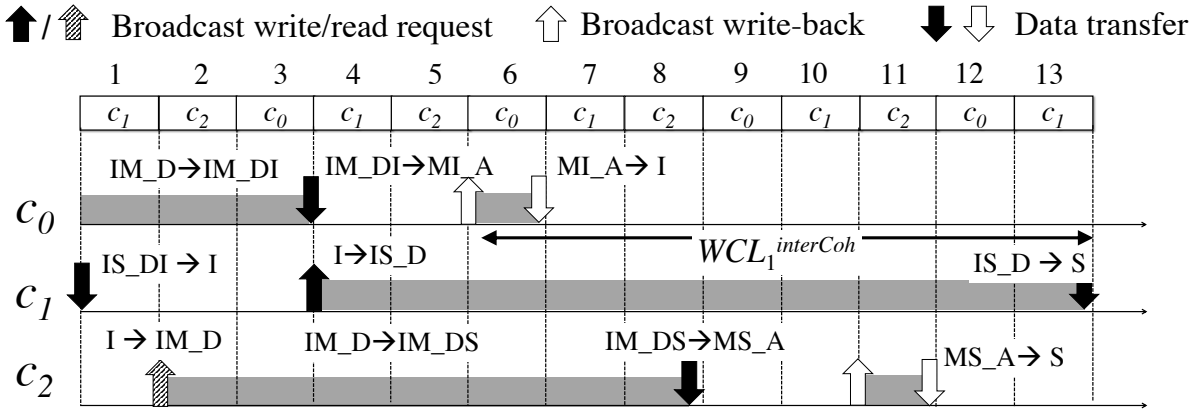


Figure 3.9: WC inter-core coherence latency. c_1 is c_i.

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

Lemma 2. *For PMSI, the WC inter-core coherence latency, $WCL_i^{interCoh}$, of a request by c_i to* A *occurs when the remaining $N-1$ cores broadcast write requests to* A *before c_i's request.*

Proof. In PMSI, the modified data copy in the owner's cache must first be written back to memory, and the memory sends the updated data to the requesting core (Table 3.2). According to Invariant 2, the shared memory services multiple requests to the same line in order of requests observed by the shared memory. Thus, in the WC, c_i has to wait until previously pending requests to A complete and the shared memory has the updated value of A before it receives A. As a result, c_i suffers $WCL_i^{interCoh}$ when all other $N-1$ cores in the system requested to modify A before c_i issued its request. There are 3 cases. For each case, assume that each core consumes T periods to obtain A, write to it, and update the shared memory with the new value. The first case is when at least one core other than c_i does a read request to A rather than a write request. The second case is when c_i requests A before at least one core among $N-1$ cores requests A. The third case is when less than $N-1$ cores request A before c_i.

Case 1. Assume that core c_j in the remaining $(N-1)$ cores broadcasts a read request to A before c_i. When c_j receives A, it does not perform a write-back to shared memory as it does not modify A. As a result, c_i does not incur T periods from c_j's access to A. Hence, the WCL of c_i is less than $(N-1) \times T$ cycles.

Case 2. Let c_i's write request to A be broadcasted before c_j's request where c_j is in the remaining $(N-1)$ cores. Invariants 2 and 3 mandate that both cores and the shared memory respond to requests in the arrival order of requests. Hence, c_i's request is serviced before c_j, and is not affected by c_j's request to A. As a result, inter-core coherence latency of $c_i < (N-1) \times T$ periods.

Case 3. Let $N' < N-1$ cores broadcast write requests to A before c_i. As a result, c_i incurs inter-core coherence latency of $N' \times T$ cycles. This inter-core coherence latency is less than $(N-1) \times T$ cycles, and hence, this scenario cannot be the WC. □

Illustrative example. Figure 3.9 shows an illustrative example of the $WCL_1^{interCoh}$. Initially, cores c_1 and c_0 have pending read and write requests to A in this order. At ❶, c_1 receives A and completes its read request. At ❷, c_2 broadcasts a write request to A. c_0 observes the remote write request, and moves to t-state **MI_A** to mark A for write-back. c_0 receives A in ❸ and issues the write-back, which is completed in its slot at ❻. Before c_0 can complete its write-back, c_1 broadcasts a read request to A at ❹. Hence, c_1 must wait for both c_0 and c_2 to complete write-backs to A before receiving A. After c_0 completes its write-back, c_2 receives A in ❽, and schedules the write-back of A in ⓫. c_1 receives A at ⓭ resulting in $WCL_1^{interCoh}$ = 10 slots.

Lemma 3. *For PMESI, the WC inter-core coherence latency, $WCL_i^{interCoh}$, for a request by c_i to* A *occurs in one of the following scenarios (1) remaining $N-1$ cores broadcast write requests to* A *before c_i's request or (2) from the remaining $N-1$ cores, a core broadcasts a read request to* A*, and then the remaining $N-2$ cores broadcast store requests to* A *before c_i's request.*

Proof. In PMESI, lines in **E** or **M** state must be written back to shared memory on observing remote memory activity on the same line (Table 3.2). Hence, two worst-case scenarios exists for PMESI that result in the worst-case inter-core coherence latency. The first worst-case scenario is similar to PMSI (Lemma 2) where the remaining $N-1$ cores broadcast write requests to the same line before c_i's request. The second worst-case scenario occurs when core c_j in the remaining $N-1$ cores first broadcasts a read request, and then the remaining $N-2$ cores broadcast write requests to A before c_i's request to A. In this scenario, core c_j receives A in **E** state as no other core has A in their private caches. Recall from Section 2.2 that only one core can have a line in **E** state. On observing remote write requests from other cores, c_j marks A for write-back, and completes the write-back in the next allocated slot. Since the remaining $N-2$ cores perform write operations, each of the $N-2$ cores must first complete the store operation, and then write-back A. We omit a detailed proof regarding this scenario as it is similar to the proof of Lemma 2. □

Lemma 4. *For Opt-PMESI, the WC inter-core coherence latency, $WCL_i^{interCoh}$, for a request by c_i to* A *occurs when the remaining $N-1$ cores broadcast write requests to* A *before c_i's request.*

Proof. Recall from Section 3.6.2, cores do not perform write-back responses for lines in **E** state in Opt-PMESI. As a result, only lines in **M** state trigger write-back responses in Opt-PMESI on remote memory activity and cache line replacements. Hence, the worst-case scenario for Opt-PMESI is equivalent to PMSI, and the proof is similar to that of Lemma 2. □

Lemma 5. *For PMSI, PMESI, and Opt-PMESI, $WCL_i^{interCoh}$ is calculated by Equation 3.2.*

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

Proof. From Lemma 2, c_i has to wait in WC for $N-1$ cores to obtain the line from the memory, perform the write operation, and finally update the shared memory with the new value. In WC, this procedure consumes two TDM periods for each other core, which leads to a total of $2(N-1)$ TDM periods. This accounts for the first component in Equation 3.2. Figure 3.9 shows the WC inter-core coherence latency for c_1 in a three-core system, where c_1 waits for 4 periods from the stamp of issuing the request to the bus until its data is ready to be sent by the memory. Moreover, if $N > 2$, when the shared memory has the updated version that is ready to send to c_i, c_i might have missed its slot in the current period. Therefore, it has to wait for an additional period to be able to receive A from the shared memory. In Figure 3.9, the WC inter-core coherence latency of c_1 is 5 TDM periods, i.e., 15 slots. On the other hand, if $N \leq 2$, the core is guaranteed to have a slot in the same period as the data is ready at the memory. This accounts for the second component in Equation 3.2. Recall that each TDM period is $N \cdot S$ cycles. $WCL_i^{interCoh}$ is as calculated by Equation 3.2. □

Lemma 6. *For PMSI, PMESI, and Opt-PMESI, the WC intra-core coherence latency is calculated by Equation 3.3, where N is the number of cores and S is the TDM slot width in cycles.*

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

Proof. There exist two cases:

Case of $N > 2$. A request from c_i implies two actions from c_i. First, issuing the request to the bus. Second, receiving the data from the shared memory. As a result, the worst-case intra-coherence latency occurs when each of these actions is delayed by write back responses that c_i has to conduct. Since the system deploys a work-conserving TDM between responses and own requests. Each action can encounter a maximum delay of one TDM period. Accordingly, the WC intra-coherence latency is two TDM periods or $2 \cdot N \cdot S$.

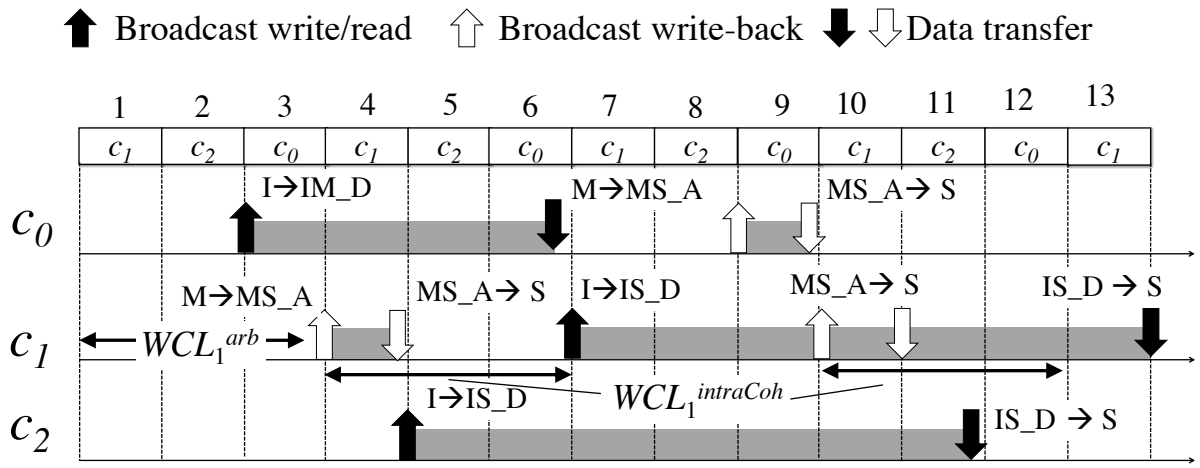


Figure 3.10: WC intra-core coherence latency. c_1 is c_i.

Case of $N \leq 2$. Recall that each core can have at maximum one pending request at any instance. Hence, c_i cannot have two pending write back requests from the only other core in the system, c_j. In worst-case, c_i requests a line that is modified by c_j. Thus, it has to wait for two TDM periods because of inter-core coherence interference as per Lemma 5. In addition, c_i can have a worst-case arbitration latency of one TDM period as per Lemma 1. During this delay, which is three TDM periods at worst, c_i can have up to only one pending write back. This is because of the TDM arbitration between write backs and own requests. □

Illustrative example. Figure 3.10 shows the $WCL_1^{intraCoh}$. c_1 generates a request to A immediately after the start of ❶. As a result, it cannot broadcast its request in ❶, and waits for its next slot (❹). However, ❹ is designated as a write-back slot, and c_1 completes a pending write-back in this slot. Thus, c_1 does not broadcast its request to A until ❸, which is one TDM period later. c_1 must wait for c_0 to write-back the updated value of A before receiving A. c_0 writes back A in ❾, and the shared memory can send A to c_1 in ❿. However, ❿ is designated as a write-back slot, and c_1 completes another pending write-back in this slot. Hence, c_1 receives A in ⓭, which is one TDM period later. The delay experienced by c_1 in broadcasting its request and receiving A is the intra-core coherence latency, which is shown in Figure 3.10.

Theorem 1. *The total WCL suffered by a core c_i issuing a request to a shared line A under PMSI, PMESI, and Opt-PMESI is calculated as:*

$$WCL_i^{tot} = WCL_i^{arb} + WCL_i^{interCoh} + WCL_i^{intraCoh} + L^{acc} \tag{3.4}$$

Proof. The total WCL is the sum of the latency components: arbitration, inter- and intra-coherence, and the access latencies. □

Coverage of unpredictability sources. Section 3.5 listed five unpredictability sources and their associated design invariants. Missing an unpredictability source means that the WCL bound is larger than the one the above analysis provides. We argue that we covered all possible unpredictability sources.

Lemma 7. *Section 3.5 cover all possible unpredictable scenarios under which a core's memory request can suffer from unbounded latency.*

Proof. Assume that there exists an unaccounted unpredictable source. As a result, this unaccounted unpredictable source will result in a WCL greater than that derived in Theorem 1. The worst-case scenario consists of memory requests across cores to data that result in the WCL. To construct the worst-case scenario, we categorize a core's (c_i) memory request into three categories: (1) the memory request is a *cache hit*, (2) the memory request is *not* a cache hit and the requested data is *neither* simultaneously cached in other cores' caches nor simultaneously requested by other cores, and (3) the memory request is *not* a cache hit and the requested data is *either* simultaneously cached in at least another core's cache or simultaneously requested by at least another core. Memory requests in (1) do not access the bus to broadcast coherence messages or wait for the requested data. This is because the requested data is available in c_i's private cache, and the memory request can operate on the cached data. As a result, the WCL_i^{arb}, $WCL_i^{interCoh}$, and $WCL_i^{intraCoh}$ are 0. Hence, memory requests in category (1) cannot constitute the worst-case scenario. Memory requests in (2) access the bus to broadcast coherence messages and wait for the requested data. However, since other cores neither simultaneously cache nor make requests to the same data, $WCL_i^{interCoh}$ is 0. Hence, the worst-case scenario cannot consist of memory requests in (2). Memory requests in (3) also access the bus to broadcast coherence messages and wait for the requested data. Furthermore, the requested data is either simultaneously cached in another cores' caches or simultaneously requested by other cores. Hence, in the worst-case, c_i must wait for other cores' to complete their requests and perform any actions (write-backs) before receiving the requested data. Therefore, WCL_i^{arb}, $WCL_i^{interCoh}$, and $WCL_i^{intraCoh}$ are $\neq$ 0, and the worst-case scenario must consist of memory requests in (3). However, the scenarios described in the analysis in Section 3.7 do indeed consist of memory requests that fall in (3). Furthermore, these scenarios are the worst-case scenarios that result in the WCL. Since the latency analysis are for protocols that satisfy the design invariants listed in Section 3.5, we have covered all possible unpredictable scenarios. □

Theorem 2. *A cache coherence mechanism is timing predictable if and only if it satisfies the design invariants listed in Section 3.5*

Proof. **Proof for $\rightarrow$ (necessary condition).** Let CC be a cache coherence mechanism that is timing predictable. This means that there does not exist a scenario under CC where a memory

request from a core has unbounded latency. From Lemma 7, this means the implementation of CC does not exhibit the scenarios listed in Section 3.5. Since each scenario listed in Section 3.5 has a corresponding design invariant, CC's implementation satisfies all the design invariants listed in Section 3.5.

Proof for $\leftarrow$ (sufficient condition). Let CC satisfy the design invariants listed in Section 3.5. CC satisfies the design invariants either at the cache coherence protocol level, micro-architecture, or a combination of both. Each design invariant in Section 3.5 fix a scenario where a core can have unbounded latency when multiple cores simultaneously cache coherent data in their private caches. Since Lemma 7 proves that Section 3.5 covers all possible unbounded latency scenarios, satisfying all the corresponding design invariants in Section 3.5 ensures that there does not exist a scenario where a core's memory request can have unbounded latency. Hence, CC is timing predictable. □

3.8 Evaluation

We integrate PMSI, PMESI, and Opt-PMESI into the gem5 simulator [17]. We use the Ruby memory model in gem5, which is a cycle-accurate model with a detailed implementation of cache coherence events. We use a multi-core architecture that consists of in-order x86 cores running at 2GHz. Each core has a private 16KB direct-mapped L1 cache, with its access latency as 3 cycles. All cores share an 8-way set-associative 1MB LLC cache. Since the focus of this work is on coherence interference, we use a perfect LLC cache to avoid extra delays from accessing off-chip DRAM. Consequently, the access latency to the LLC is fixed, and equals to 50 cycles ($L^{acc} = 50$ cycles). The DRAM access overheads can be computed using other approaches such as [64, 159], and they are additive [165] to the latencies derived in this work. Both L1 and LLC have a cache line size of 64 bytes. The interconnect bus uses TDM arbitration amongst cores. The L1 cache controller uses work-conserving TDM arbitration between a core's own requests and its responses to other core requests. We do not run an operating system in the simulator, and hence, all memory addresses generated by the cores are physical memory addresses. We evaluate PMSI, PMESI, and Opt-PMESI using the *SPLASH-2* [157] benchmark suite. In addition, we use synthetic workloads to stress the WC behavior.

3.8.1 Verification

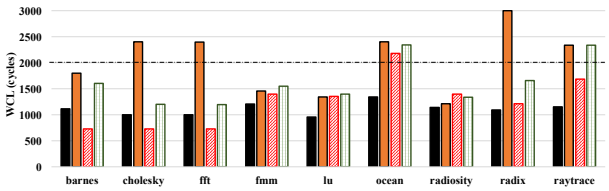
We verified the correctness of PMSI, PMESI, and Opt-PMESI using various methods. 1) We used the Ruby Random Tester with gem5 [17] specifically to verify coherence protocols. We

stressed all protocols with 10 million random requests. 2) We used carefully-crafted synthetic micro-benchmarks to cover all possible transitions and states all the three protocols. This also ensures the exhaustiveness of the identified unpredictability sources and corresponding invariants. If there was an unpredictability source that is not included in Section 3.5, it should lead to unpredictable behavior (i.e., observed latencies would exceed the bound) at one or more of the transitions, which we did not observe. 3) We executed the applications in the SPLASH-2 suite on gem5 using PMSI, PMESI, and Opt-PMESI and they run to completion. Furthermore, we check data correctness by checking the output of each application. 4) We also formally verified the correctness properties and WCL bounds for the protocols using the formal models developed by Sensfelder et al. [133].

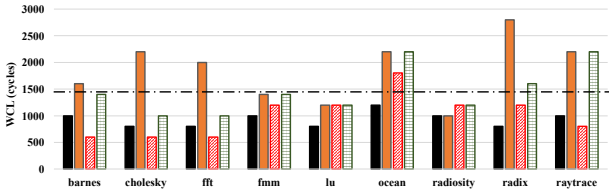
3.8.2 Observed worst-case latencies

We study the effectiveness of PMSI, PMESI, and Opt-PMESI to bound the delays resulting from coherence interference. We also study the effects of violating each one of the invariants on the memory latency. We use a 4-core system for our experiments. For SPLASH-2, we launch each SPLASH-2 application as four threads using four single-threaded cores, where only one application is used per experiment. Figure 3.11 depicts our findings, and shows the total observed memory latency. Since SPLASH-2 applications are optimized to minimize data sharing, they do not stress the coherence protocol. Therefore, to further stress the coherence protocol, we execute synthetic experiments using 9 synthetically-generated workloads: Synth1 to Synth9 in Figure 3.11. In each synthetic experiment, we simultaneously run four identical instances of one workload by assigning one instance on each core. These experiments represent the maximum possible sharing of data since each core generates the same sequence of memory requests. The WC arbitration latency for benchmarks in all experiments is $N \cdot S = 200$ cycles for $N = 4$ cores and slot $S = L^{acc} = 50$ cycles; hence, not shown. Since all three protocols have the same worst-case scenarios and latencies (Section 3.7), we present results only for the PMSI protocol. We verified that the below observations also apply to PMESI and Opt-PMESI. Figure 3.12 shows the violin plot distributions of memory request latency under different predictable cache coherence protocols.

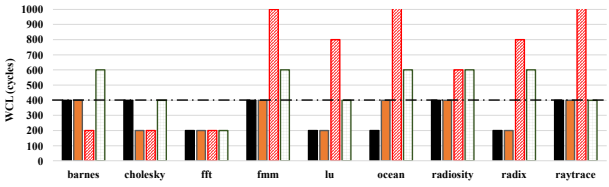
Observations. (1) Figure 3.11 shows that for PMSI the total WC latencies are within their analytical total WCL bounds. We also observed that the individual latency components such as the arbitration, inter-core, and intra-core coherence latency components are within their respective analytical WCL bounds derived in Section 3.7. This is shown Figure 3.11. (2) On the other hand, violating any of the invariants introduces a source of unpredictability, which results in exceeding those bounds. Moreover, for source 1, one of the cores is not able to obtain an access to a block that it requests and the program never terminates. This is the reason that Figure 3.11 does not



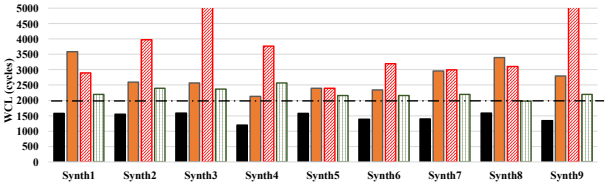
(a) Total memory latency for SPLASH-2 suite.



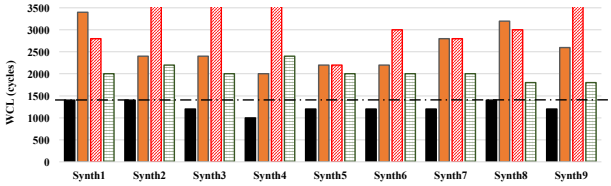
(b) Inter-core coherence latency for SPLASH-2 suite.



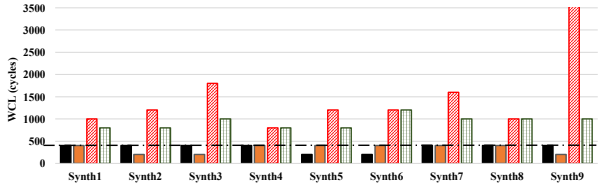
(c) Intra-core coherence latency for SPLASH-2 suite.



(d) Total memory latency for synthetic workloads.



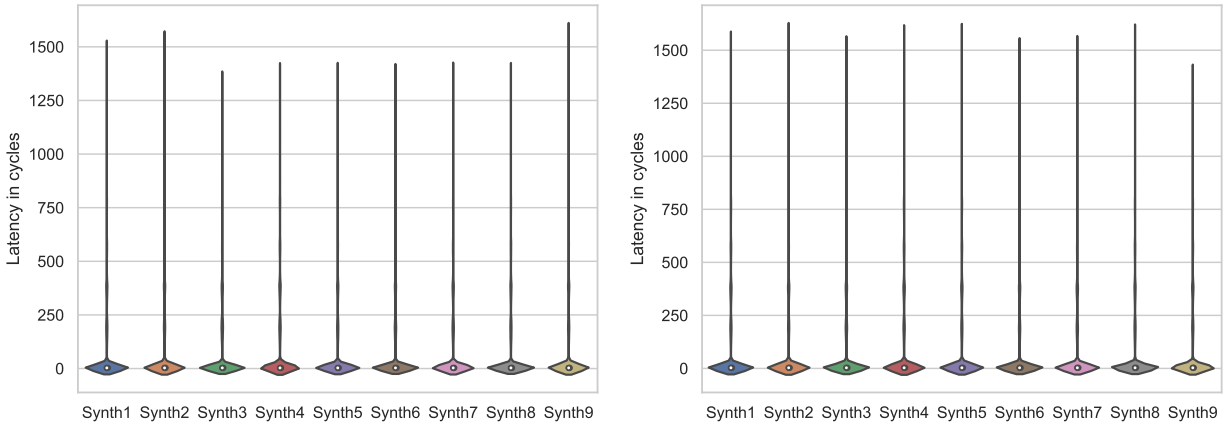
(e) Inter-core coherence latency for the synthetic workloads.



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

Figure 3.11: WC latencies and the effect of unpredictability sources on them. Horizontal dotted line represents the analytical bound. Black bars are PMSI, PMESI, and Opt-PMESI protocols, orange bars denote violating design invariant 2, red bars denote violating design invariant 3, and green bars denote violating design invariant 4.

show Unpredictable 1. This shows that augmenting a conventional coherence protocol with a predictable arbiter does not guarantee predictability. Note that violating some of the invariants also results in exceeding the latency bounds of the individual latency components. For example, we observed that violating invariant 3 causes resulted in the observed inter-core coherence latency to exceed the corresponding analytical bound across all synthetic and SPLASH-2 benchmarks. (3) For a quad-core system, the latency suffered by a core due to coherence interference is $9\times$ more than the latency due to bus arbitration. The inter-core coherence interference solely



(a) PMSI on synthetic workloads.

(b) PMESI on synthetic workloads.

Figure 3.12: Memory request latency distribution under PMSI and PMESI protocols.

contributes a latency up to 7× of the arbitration latency, while the latency resulting from the intra-core coherence interference is double the arbitration latency. This provides evidence of the importance of considering the coherence latency when sharing data across multiple cores for real-time applications. (4) From the violin plot distributions in Figure 3.12, most memory requests to shared data under PMSI, PMESI, and Opt-PMESI protocols benefit from caching, and experience lower memory request latency. This highlights the key benefit of using predictable cache coherence protocols compared to alternative predictable shared data mechanisms that constrain private caching of shared data. The maximum observed memory request latency across synthetic benchmarks under PMSI, PMESI, and Opt-PMESI are within the analytical WCL bound.

3.8.3 Comparison against prior predictable approaches

We compare the overhead caused by four alternative predictable approaches to handle data sharing in multi-core real-time systems: (1) not using private caches (*uncache-all*), (2) not caching the shared data (*uncache-shared*), (3) the proposed PMSI, PMESI, and Opt-PMESI protocols, and (4) mapping all tasks that share data to the same core (*single-core*). For the first three approaches, each application is distributed across four-cores. *uncache-shared* is an adaptation of the approach by [61, 86], but for data instead of instructions. *single-core* maps tasks with shared data to the same core to eliminate incoherence due to shared data, which adopts the idea of data-aware scheduling [23]. The overhead is calculated as the slowdown compared to the conventional MESI protocol. Figure 3.13 depicts our findings for the SPLASH-2 workloads, where MSI and MESI are the conventional (unpredictable) protocols implemented as in [138].

Observations. (1) Across all benchmarks, the *uncache-all* has the worst execution time with a

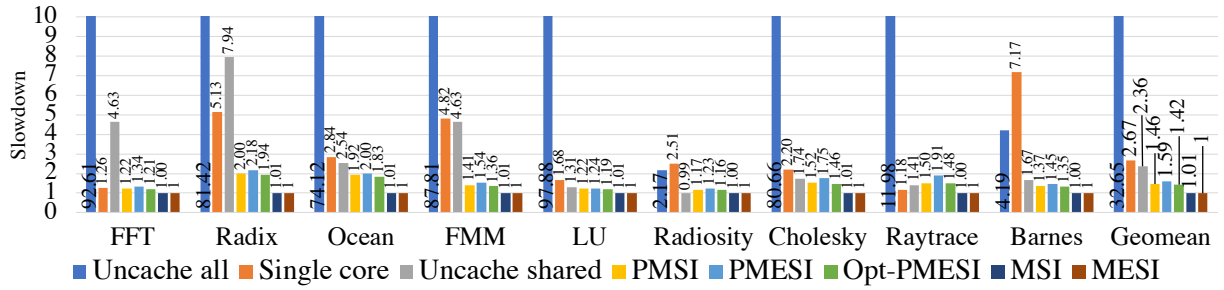
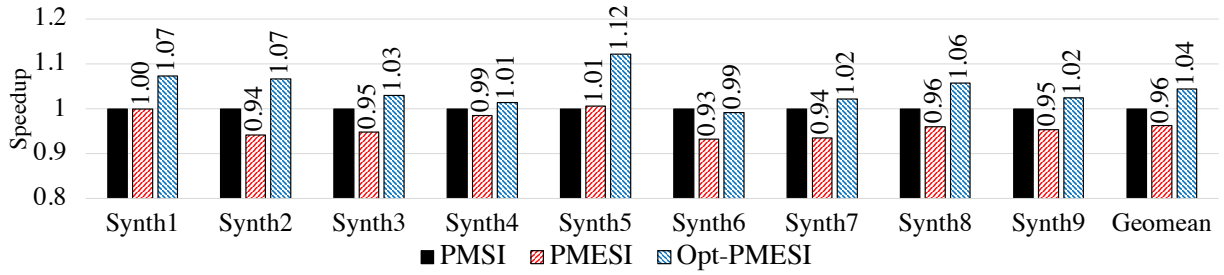
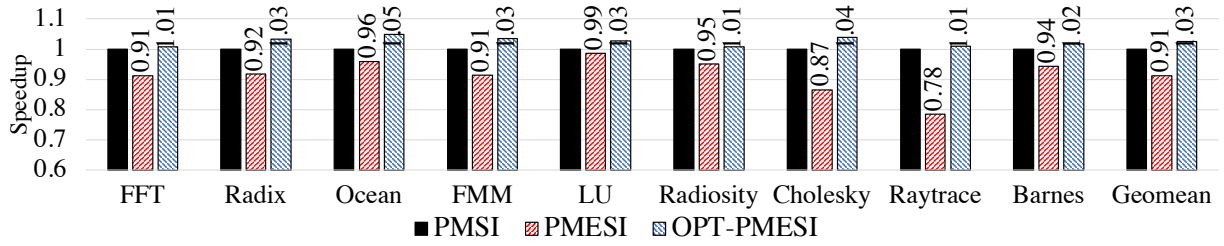


Figure 3.13: Execution time slowdown compared to MESI protocol.

geometric mean slowdown of $32.66\times$ compared to MESI, followed by *single-core* and *uncache-shared* with geometric mean slowdowns of $2.67\times$ and $2.11\times$ respectively. The *uncache-shared* and *single-core* approaches require additional hardware and software modifications to the applications and RTOS to track cache lines shared between cores. (2) Since private data does not cause any coherence interference, *uncache-shared* allows caching of only private data, while uncaching of all shared data. In Figure 3.13, *uncache-shared* has better performance than *uncache-all* for all applications with a geometric mean slowdown of $2.11\times$ Nonetheless, *uncache-shared* requires additional hardware and software modifications to distinguish and track cache lines with shared data, which are the same modifications required by [115]. (3) Mapping applications with shared data to the same core avoids data incoherence since these tasks share the same private cache. However, it prohibits parallel execution of these application. In consequence, for some applications (*fft*, *radix*, and *raytrace*), *single-core* achieves better performance compared to *uncache-shared*, while for other applications, it exhibits lower performance. This is dependent on several factors such as the memory-intensity of the application and the ratio of shared to non-shared data. Overall, *single-core* achieves a geometric slowdown of $2.67\times$. (4) On the other hand, PMSI, PMESI, and Opt-PMESI protocols achieve better performance compared to all other predictable approaches with no changes to the application or RTOS. PMSI, PMESI, and Opt-PMESI achieve improved performance of up to $4\times$ the best competitive approach, *uncache-shared*, with a geometric mean slowdown of $1.46\times$, $1.59\times$, and $1.42\times$ in performance compared to MESI respectively. (3) For the synthetic benchmarks, the *single-core* approach offers $2.9\times$ average performance speedup over *uncache-shared* approach. This is because the *uncache-shared* approach disallows any caching of memory addresses, and hence, no memory requests result in cache hits. The PMSI, PMESI, and Opt-PMESI protocols offer $3.08\times$, $2.99\times$, and $3.12\times$ average performance speedup over *uncache-shared* respectively. Compared to *single-core*, PMSI, PMESI, and Opt-PMESI exhibit performance speedups as high as 16% without constraining core utilization.



(a) Synthetic benchmarks.



(b) SPLASH-2 benchmark suite.

Figure 3.14: Average-case performance speedups of PMSI, PMESI, and Opt-PMESI for synthetic and SPLASH-2 benchmarks.

3.8.4 Comparison of PMSI, PMESI, and Opt-PMESI protocols

Figures 3.14a and 3.14b compare the average-case performance of the PMSI, PMESI, and Opt-PMESI protocols for the synthetic and SPLASH-2 benchmarks respectively against PMSI. While all the protocols have the same WCL bounds, PMESI and Opt-PMESI have additional states and transitions that enable average-case performance improvements over PMSI. Figures 3.14a and 3.14b normalize the average-case performance to PMSI.

Observations. (1) Across synthetic and SPLASH-2 benchmarks, PMESI does not provide performance benefits over PMSI. The key reason for this is the increased number of write-backs in PMESI due to states **E** and **M**. In PMESI, a core triggers a write-back for a line that it has in **E** or **M** state. Recall from Section 3.6.1 that cores deploy a predictable arbitration scheme that services write-backs and pending demand requests in a core's allocated slot. As a result, the increased number of write-backs in PMESI contend for the allocated slots resulting in longer execution time to complete cores' demand requests. For the synthetic benchmarks, we observed that PMESI experiences 24% more write-backs on average than PMSI. For the SPLASH-2 benchmarks, PMESI experiences 1.9× more write-backs than PMSI. This is because the working data set sizes of the SPLASH-2 benchmarks do not fit in the private caches resulting in more cache line evictions due to capacity misses. As a result, in PMESI, 44% of the total write-backs in SPLASH-2 are due to cache line evictions to lines in **E** state. Hence, PMESI does not improve

over PMSI (4% average performance degradation for synthetic benchmarks and 9% average performance degradation for SPLASH-2 benchmarks) due to the increased number of write-back responses that contend for allocated slots with demand requests. (2) The additional hardware overhead in Opt-PMESI addresses this performance limitation of PMESI, and improves over PMSI and PMESI for both the synthetic and SPLASH-2 benchmarks. For synthetic and SPLASH-2 benchmarks, Opt-PMESI improves performance by 4% and 3% respectively. The performance improvement is primarily due to silent writes that allows cores to complete writes on lines in **E** state without broadcasting on the bus.

3.9 Conclusion

We point out possible sources of unpredictable behavior in conventional coherence protocols. To address this unpredictability, we describe a set of invariants. These invariants are general and can be applied to other coherence protocols. We show how to deploy these invariants in the fundamental MSI and MESI protocols. Towards this target, we propose a set of novel transient states as well as minimal architecture requirements resulting in predictable MSI (PMSI) and predictable MESI (PMESI) protocols. Furthermore, we design Opt-PMESI, an alternative protocol that addresses the performance limitations of PMESI. We derive WCL bounds for all three protocols, and experiment using the SPLASH-2 benchmark suite and worst-case oriented synthetic workloads. Our evaluation shows that (1) the invariants implemented in all three protocols ensure that that the observed WC latencies are within the derived analytical bounds, and (2) the average-case performance of our approaches offer significant average-case performance over state-of-the-art predictable approaches for shared data accesses.

Chapter 4

Balancing Predictability and High-Performance in Cache Coherence Mechanisms

The *second research contribution* of this thesis brings to attention the *WCL gap* between predictable cache coherence mechanisms and prior predictable shared data communication mechanisms. This WCL gap makes predictable cache coherence an *inferior* communication choice compared to prior predictable shared data communication mechanisms due to their relatively high WCL. While the tools from the previous chapter (Chapter 3), in the form of design invariants, facilitate the design of predictable hardware cache coherence mechanisms, they do not provide any guidance on how to improve their timing predictability. Improving the timing predictability means reducing the WCL of a memory request. To this end, we present a systematic framework that captures the relationship between predictable cache coherence mechanism design and their corresponding WCL. Using this framework, we present a technique to reduce the WCL under predictable cache coherence mechanisms through changes to the underlying cache coherence protocol. Our design technique results in predictable cache coherence mechanisms that have the same WCL guarantees as prior communication mechanisms, thereby *eliminating* the WCL gap, while still maintaining the performance advantage over prior mechanisms.

4.1 Introduction

Satisfying temporal properties of safety-critical tasks in the form of worst-case latency bounds (WCL) is the *paramount* consideration; average-case performance is typically a secondary con-

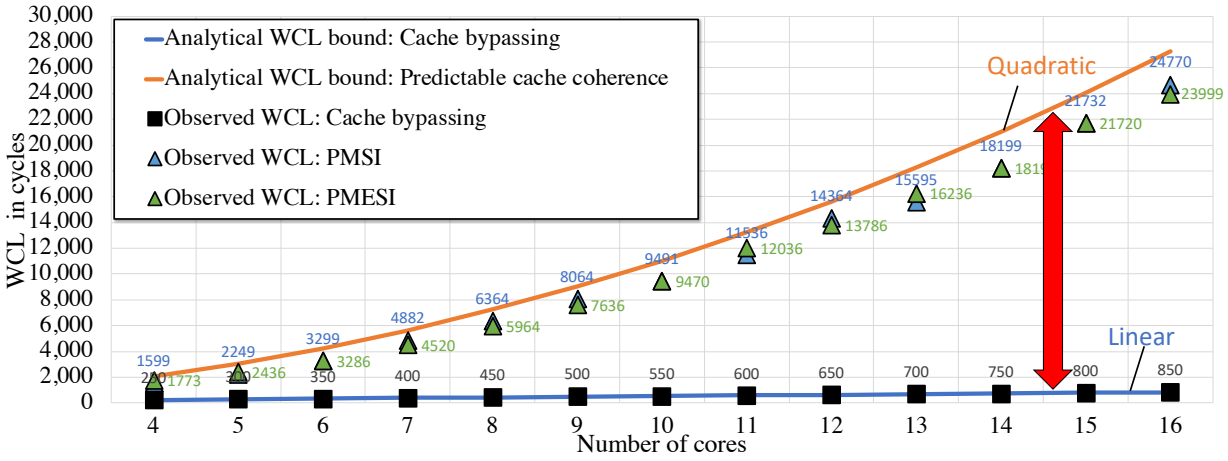


Figure 4.1: Variation of WCL for alternative and predictable cache coherence mechanisms with core count on synthetic workloads.

sideration. This is because WCL bounds are used to determine an execution schedule of safety-critical tasks, and tighter WCL bounds improves task schedulability. This means that a data communication mechanism that offers large WCL bounds is an *unacceptable* mechanism irrespective of the stellar average-case performance it provides. Based on this criterion, Figure 4.1 shows that predictable cache coherence mechanisms proposed so far [74, 77, 139] are *unacceptable* data communication mechanisms.

Figure 4.1 plots the analytical WCL bound and observed WCL for different data communication mechanisms as a function of the number of cores (N). We take mechanisms described in prior works such as cache bypassing of shared data [25, 61, 86], and the PMSI and PMESI cache coherence protocols [74]. Cache bypassing mechanism disables cores from caching shared data, and hence, sidesteps cache coherence by forcing all shared data to reside in the shared memory. As a result, the WCL of a memory request under cache bypassing is the latency to access the shared memory, which grows *linearly* with the N. On the other hand, prior works on predictable cache coherence mechanisms [74, 77, 139] showed that WCL of a memory request grows *quadratically* with N. To put these trends into perspective, we take a 8-core platform; a core count of 8 cores is representative of multi-core real-time platforms such as the NXP QorIQ T4 platform [106]. We will assume a TDM-based shared bus arbitration where each core receives a time slot of length 50 cycles. For a 8-core system, the analytical WCL bound of existing predictable cache coherence protocols is **7250** cycles, and the analytical WCL bound of the cache bypassing mechanism is **450** cycles. Our evaluation shows that the 16× WCL gap returns at most a 5× performance speedup over cache bypassing for the SPLASH-2 benchmark. We argue that this *disparity* between WCL gap and performance returns is not acceptable, and brings into question the applicability and possibly the commercial adoption of predictable cache co-

herence mechanisms for safety-critical platforms with a reasonable number of cores. Rather, a 4× performance speedup over cache bypassing and with *same* WCL (no WCL gap) would make predictable cache coherence mechanisms more compelling and acceptable data communication mechanisms.

Unfortunately, recent prior works such as [13, 65] that address this WCL gap in predictable cache coherence mechanisms do not provide a *systematic* approach to design predictable cache coherence mechanisms with tight WCL *and* high-performance. These works provide *ad hoc* approaches that focus on a *specific* cache coherence protocol and enforce caching *constraints* to bridge the WCL gap. On the other hand, a systematic approach provides a designer the key reasons behind the high WCL in predictable cache coherence mechanisms, which then allows the designers to reason about approaches to designing coherence mechanisms that achieve tight WCL and high-performance.

In this work, we present a *systematic approach* towards designing predictable cache coherence mechanisms that offer tight WCL and high-performance. Tight WCL means that the WCL of a memory request grows linearly with core count. The resulting cache coherence mechanisms derived on applying our approach have the *same* WCL as the cache bypassing mechanism thereby *bridging* the WCL gap. For example, we design predictable cache coherence mechanisms with WCL of 450 cycles for a 8-core system, which is a 94% reduction compared to the WCL of existing predictable cache coherence mechanisms. For an approach to be systematic, it must be guided by some formal analysis that captures the root causes of the high WCL in existing predictable cache coherence mechanisms. To this end, our first main contribution is a *formal framework* and *analysis* that identifies the design features of a cache coherence mechanism that contribute to its WCL. Our second contribution describes one technique towards designing predictable cache coherence mechanisms that offers tight WCL and high-performance through micro-architectural extensions and cache coherence protocol changes. This technique is guided by the insights derived from our formal framework and analysis, and hence, systematic. Our design technique distinguishes itself from prior works [13, 65] in that it does *not* apply any caching constraints, and can be applied to tighten the WCL of *different* cache coherence protocols. We apply our design technique to two cache coherence protocols, and show that the resulting protocols maintain their performance advantage over the cache bypassing mechanism (up to 5× performance speedup) while having the same WCL as the cache bypassing mechanism.

4.2 Main contributions

In summary, the main contributions of this work are:

1. A systematic approach that consists of a formal framework to model and analyze predictable cache coherence mechanisms and their WCL. This framework identifies the root

causes for the high WCL in existing predictable cache coherence mechanisms (Section 4.6).

2. Guided by the analysis, we focus on one technique to design predictable cache coherence mechanisms with tight WCL and high-performance. We show that the resulting cache coherence mechanisms from applying this technique have the same WCL as that of cache bypassing, thereby *eliminating* the WCL gap (Section 4.8).

3. We evaluate two cache coherence mechanisms obtained from applying our proposed technique using the gem5 micro-architectural simulator [17]. Our performance evaluation shows that the new mechanisms maintain their performance advantage over cache bypassing (up to 5× performance speedup) and sacrifice at most 13% of the original coherence mechanisms (Section 4.9).

4.3 Related work

Hassan et al. [74] presented a template for designing predictable cache coherence protocols, and proposed the PMSI protocol using the design template. The template defined design invariants for designing predictable cache coherence protocols. In this work, all the protocols including PMSI*, and PMESI*, satisfy the design invariants described in [74]. Sritharan et al. [139] designed Pendulum, a time-based predictable cache coherence protocol for mixed-critical multi-core systems wherein cores retained cache lines for specific time duration based on the criticality level of the tasks executing on the cores. State transitions in Pendulum were triggered on memory activity and physical time, and some coherence states in Pendulum encoded information about physical time. Our formal framework does not capture physical time, and hence, cannot model the coherence states and transitions in Pendulum. Kaushik et al. [77] designed CARP, a PMSI-based predictable cache coherence protocol for mixed-critical multi-core systems. CARP allowed data communication between non-critical and critical cores while disallowing timing interference from non-critical tasks on critical cores [77]. CARP can be modeled using our formal framework, and the changes to the PMSI protocol in Section 4.8 can tighten the WCL bound of memory requests from critical cores under CARP. Bansal et al. [13] and Hassan [65] presented predictable cache coherence protocols that tightened the WCL under predictable cache coherence mechanisms through a combination of specific protocol changes and caching constraints. The predictable cache coherence protocol in [13] disallowed communicated data to reside in the core's private caches, and the protocol in [65] disallowed modified communicated data to reside in the cores' private caches. These protocols required support from the software application to distinguish between data that is communicated between cores and private to a core. On the other hand, our technique of protocol changes along with point-to-point data interconnects achieves

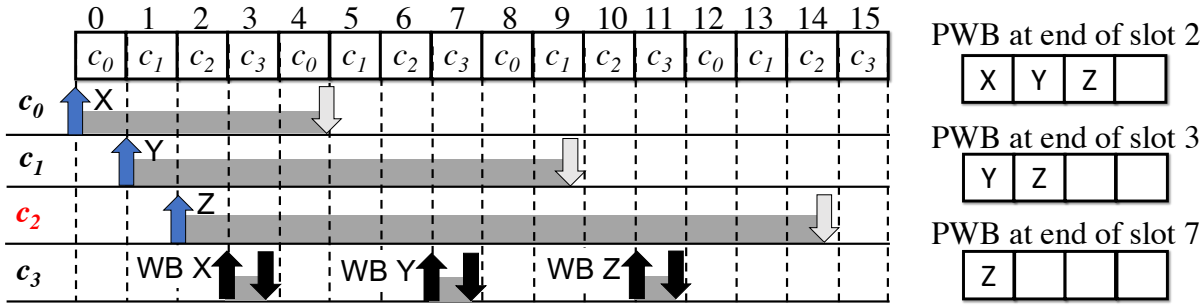


Figure 4.2: Example execution under PMSI protocol.

tight WCL and high-performance without placing any caching constraints. The transformed protocols in Section 4.8 have the same WCL bound as [13, 65].

Recently, Salah and Hassan [67] described a new bus architecture for snooping bus-based cache coherence mechanisms that tightened the WCL of a memory request. The key insight behind their bus architecture is the decoupling of coherence message communication with cores and shared memory and the data communication between the cores and shared memory. This decoupling enables achieves tighter WCL bounds compared to prior predictable cache coherence mechanisms. This work is complimentary to [67] and achieves tighter WCL bounds through protocol changes while preserving the conventional predictable bus architecture that couple message and data communication. The WCL of a memory request under our proposed approach is the same as that under [67].

4.4 Motivation

4.4.1 High level understanding behind the WCL gap

Bridging the WCL gap requires first understanding the *root cause* of the high WCL in existing predictable cache coherence protocols. We present a high level understanding of the WCL gap using Figure 4.2 as an illustrative example. Section 4.7 provides a formal treatment of the WCL gap analysis. Figure 4.2 shows an execution using the PMSI coherence protocol on a 4-core platform. c_2 is the core under analysis. The shared bus deploys a TDM arbitration that allocates one time slot to each core that is large enough to complete one memory transaction between core and the shared memory. We use ❶ to denote TDM slot i.

Initially, c_3 has cache lines X, Y, and Z in **M** state. Cores c_0, c_1, and c_2 make read requests to X, Y, and Z at ❶, ❷, and ❸ respectively. Based on the PMSI protocol (Table 3.2 in Chapter 3), the read requests from c_0-c_2 causes c_3 to write-back the updated versions of X, Y, and Z to shared memory. On receiving the updated versions, the shared memory can send the data to the

requesting cores. Write-back operations to the shared memory require access to the shared data bus. As a result, c_3 has to wait for its allocated slots to complete the write-back operations. Figure 4.2 shows c_3's pending write-back buffer (PWB), which records the write-backs. c_3 completes the write-back operations at ❸, ❼, and ❿ for X, Y, and Z respectively. We make two observations pertinent to the WCL of c_2's memory request to Z.

Observation 1. c_3's data write-back operations to shared memory are *in response* to the read requests from c_0, c_1, and c_2. For example, c_3's write-back to X is in *response* to the c_0's read request to X. These data write-backs to shared memory are necessary for maintaining data correctness. Since the write-backs require access to the shared buses, c_3 must wait for its allocated time slots on the shared buses to complete the write-backs to shared memory as shown in Figure 4.2.

Observation 2. c_2 must *wait* for c_3 to complete all prior write-backs recorded before Z in its PWB. This is because a core's PWB entries are serviced in a FIFO manner [74]. As a consequence, c_2 must wait for previous write-backs (X and Y) and the write-back of the requested Z before receiving Z from the shared memory.

From these two observations, c_2 has to wait for prior $N-1$ write-backs to the shared memory before its requested data is written to shared memory. Since each write-back operation from a core waits for its allocated slot ($\mathcal{O}(N)$), the WCL of c_2's request scales quadratically with N ($\mathcal{O}(N^2)$). Hence, at a high level, the root cause of this quadratic WCL in existing predictable cache coherence protocols can be attributed to:

A scenario where at least one core responds with shared bus accesses to another core's request, and these responses must complete before the requesting core can complete its request.

4.4.2 Techniques to tighten the WCL

This high-level understanding behind the WCL gap reveals different techniques to tighten the WCL. One possible technique is to allow a core to complete its responses *immediately* on observing another core's request. This can be achieved by replacing shared buses that communicate responses from cores with *dedicated* buses for each core Another technique focuses on *eliminating* scenarios where a core responses with shared bus accesses to another core's request, which is the focus of this work. In this work, we describe one technique that applies protocol changes and makes use of *direct cache-to-cache communication* between cores through point-to-point interconnects. Note that cores still communicate coherence messages through the shared message bus and communicate data with the shared memory through the shared data bus. Direct cache-to-cache communication between cores is available in existing multi-core platforms such as ARM's snoop control unit [8] and Intel's multi-processor quick path interconnect [173]. The high-level idea behind this technique is that protocol changes *convert* scenarios where a core executes shared bus accesses in response to another core's request into *direct* communication

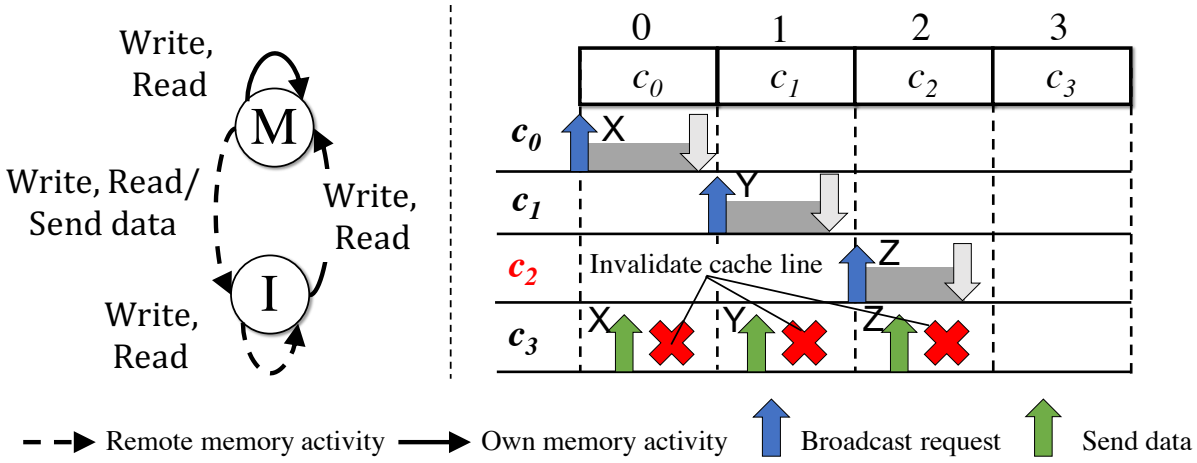


Figure 4.3: PMI protocol and execution example.

between cores. As a result, the protocol changes eliminate a core from performing shared bus accesses in response to another core's request. However, such protocol changes *may trade away significant performance opportunities* in the original protocol if not done carefully. Using the PMSI protocol as an example, we show the extent to which careless protocol changes can degrade performance. In Section 4.8, we describe one technique that refines the protocol changes to effectively balance tight WCL and average-case performance.

Consider the following protocol changes to the PMSI protocol: (1) a cache line transitions to `M` state instead of the `S` state on an own read request, and (2) a cache line in `M` transitions to `I` instead of `S` state on a remote read request. Note that these changes are valid and do not violate data correctness as a cache line in `M` has both read and write access permissions [138]. The resulting protocol due to these changes, predictable Modified-Invalid (PMI) protocol, is shown in Figure 4.3. We deploy the PMI protocol on a multi-core model that has direct cache-to-cache communication support between cores. In the PMI protocol, a core that has a cache line in `M` state sends the requested cache line to the requesting core directly and invalidates its cache line copy (move to `I` state). Applying the PMI protocol to Figure 4.2, c_3 sends the requested data to the other cores and changes the coherence states of its copies of the requested data from `M` to `I` state. These changes allow the requesting cores (c_0-c_2) to complete their requests in the same slot that it made their requests as shown in Figure 4.3.

WCL of memory request under PMI. Notice that the PMI protocol does not have any transitions where a core performs shared bus accesses in response to another core's request; for example, there are no transitions that trigger write-backs to the shared memory due to another core's request. The key reason for this lies in the following observation regarding the PMI pro-

tocol. The PMI protocol allows *only one core to have a copy of the cache line*. If one were to look at the overall state of the requested cache line across all cores before and after a memory request from the shared memory perspective, then the state of the cache line remains the *same*; there exists one core that has a copy of the cache line in **M** state. As a result, no cores respond with shared bus accesses on observing another core's request. In the worst-case, a core waits for its allocated slot to communicate its request on the shared message bus, and will receive the requested in the same slot either from the shared memory or from another core. Hence, the WCL of a memory request under PMI is $\mathcal{O}(N)$, which is tighter than that under the PMSI protocol ($\mathcal{O}(N^2)$).

Average-case performance of PMI. While the PMI protocol has tight WCL, the lack of a shared state **S** prevents multiple cores from simultaneously having the same cache line in their private caches, which in turn degrades its average-case performance benefits. For an 8-core multi-core configuration, the WCL of a memory request under PMI is tighter than that under PMSI by 94% (450 cycles vs 7250 cycles). However, our evaluation shows the PMI protocol exhibits an average performance *slowdown* of 3.1× compared to the PMSI protocol, and 2× compared to cache bypassing technique. This *imbalance* between tight WCL and performance under the PMI protocol makes it an *inferior* choice as a data communication mechanism. Hence, protocol changes to cache coherence protocols targeted at tightening their WCL must be done carefully in order to retain their existing high-performance benefits. In Section 4.8, we present one technique of protocol changes that balances tight WCL and high-performance. We describe the design of the PMSI* protocol, which is derived from applying our proposed technique to the PMSI protocol. The WCL of a memory request under PMSI* is the same as that under the PMI protocol, and the PMSI* protocol exhibits at most a 22% performance slowdown compared to the PMSI protocol.

4.5 System model

We consider a multi-core model with N cores $\mathbb{C} = \{c_0, c_1, ..., c_{N-1}\}$. We denote the set of T tasks that execute on N cores as $\Gamma = \{\tau_i \ : \ i \in [0, T-1]\}$. Multiple tasks can be mapped for execution to the same core. At any instance of time, a core executes one task; multiple cores execute tasks simultaneously. We assume cores implement in-order pipelines, and each core allows for one outstanding memory request. Cores have a memory hierarchy consisting of split level one (L1) private data and instruction caches, and a shared memory. The contents of a core's data and instruction caches are managed by the cache controllers. The shared memory contains all the data required by the tasks running on the cores, and the private caches hold a subset of the tasks' data (inclusive memory hierarchy). The cores' caches are configured as write allocate write-back caches.

The multi-core model deploys a predictable snooping bus-based cache coherence protocol,

and consists of the hardware structures presented in Chapter 3. The number of entries in the pending response buffer, and pending request lookup table is set to N; at any instance of time there can be N pending requests across all cores [74]. Pending responses in a core's pending response buffer are serviced in a first-in first-out (FIFO) order [74]. Cores communicate coherence messages on the shared snooping bus interconnect, which we refer to as the *shared message bus*. Cores communicate data with the shared memory using a *shared data bus*. The message and data buses are split-transaction non-atomic buses [138].

The analysis assumes TDM shared bus arbitration, although other predictable arbitration policies are also possible. Each TDM time slot is large enough to complete one memory operation to the shared memory, which includes the time to communicate messages based on the predictable cache coherence protocol. In an arbitration period, each core is allocated a constant number of TDM slots based on their shared memory access requirements. The slot allocation for a core is independent of the slot allocation of other cores. As a result, the worst-case asymptotic arbitration latency experienced by a core to access the shared bus is $\mathcal{O}(N)$. Prior works on multi-core real-time systems have used a similar multi-core system model [47, 61, 63, 65, 66, 74, 77, 139].

4.6 Analyzing Predictable Cache Coherence Protocols

In this section, we describe our formal framework that establishes the relationship between protocol designs and their corresponding WCL. This framework guides our efforts in tightening the WCL of existing predictable cache coherence protocols in Section 4.8. In Section 4.6.1, we present the formal model of cache coherence protocols, and in Section 4.7, we present a first-order WCL analysis using this model.

4.6.1 Formal model of coherence protocols

There are three main components of our formal model: (1) coherence states, (2) transitions between coherence states, and (3) multi-core protocol view, which captures the transitions in coherence states due to memory activity on the cache line.

Coherence states

The coherence state (stable state or transient state) of a cache line in a core's private cache encodes three pieces of information that govern the core's memory activity on the cache line, and the core's responses due to memory activity from other cores on the cache line.

1. Access permissions: Access permissions denote the read and write permissions of a cache line in a core's private cache. We denote a cache line's access permissions as $\mathbb{AP} = \{read, write,$

$eread, invalid\}$. A cache line in a core's private cache has *invalid access permission* if the core cannot read or write its data contents. This means that the cache line is either not present in the core's private cache or has incorrect data. A cache line in a core's private cache has *read access permission* if the core can read its data contents. A cache line in a core's private cache has *exclusive-read access permission* ($eread$) if the core can read its data contents, and it is the only core that has the cache line in its private cache. A cache line in a core's private cache has *write access permission* if the core can read *and* write its data contents.

2. Data state: The data state component conveys relative information about the data contents of a cache line in the core's private cache and in the shared memory. We denote the data state of a cache line as $\mathbb{DS} = \{dirty, clean\}$. A cache line in a core's private cache with *dirty* data state means that the core may have modified the data contents of the cache line. Hence, the data contents of the cache line in the core's private cache are different from that in the shared memory. On the other hand, a cache line in a core's private cache with *clean* data state means that the data contents of the cache line are the same in the core's private cache and in the shared memory. Note that multiple cores cannot simultaneously have the same cache line with $dirty$ data state.

3. Data authority: The data authority component determines whether a core with a cache line responds with data to a remote core that requests the same cache line. We denote the data authority of a cache line as $\mathbb{DA} = \{active, passive\}$. A cache line in a core's private cache with *active* data authority responds with data to remote memory activity to the same cache line. On the other hand, a cache line in a core's private cache with *passive* data authority does not respond with data to remote memory activity to the same cache line. Note that at any instance of time, only one core can have a cache line with $active$ data authority or multiple cores can have a cache line with $passive$ data authority.

$\mathbb{S} = \mathbb{AP} \times \mathbb{DS} \times \mathbb{DA}$ is the set of all possible coherence states of a cache line in a core's private cache. A coherence state of a cache line with address A in core's $c_i \in \mathbb{C}$ private cache is a 3-tuple of the form $s^{c_i}_{\mathsf{A}} = (ap^{c_i}_{\mathsf{A}}, ds^{c_i}_{\mathsf{A}}, da^{c_i}_{\mathsf{A}}) \in \mathbb{S}$ where $ap^{c_i}_{\mathsf{A}} \in \mathbb{AP}$, $ds^{c_i}_{\mathsf{A}} \in \mathbb{DS}$, and $da^{c_i}_{\mathsf{A}} \in \mathbb{DA}$. The shared memory controller also maintains coherence states for the cache lines in the shared memory. Coherence state changes for a cache line at the shared memory is due to data state and data authority changes for the cache line in the private caches. Hence, our framework does not model the coherence states in the shared memory.

State transitions

Transitions between coherence states are triggered on *events* observed by the cache controllers on the shared snooping message bus. These events are caused by memory requests issued by the cores' cache controllers due to the application execution. We denote the set of memory requests on a cache line as $\mathbb{O} = \{\mathsf{ReadReq}, \mathsf{WriteReq}\}$, and we denote the set of events on a cache line as $\mathbb{E} = \{\mathsf{OwnRead}, \mathsf{OtherRead}, \mathsf{OwnWrite}, \mathsf{OtherWrite}\}$. We next describe the generation of

events from memory requests.

A core first issues a ReadReq/WriteReq to a cache line with memory address A. If A is available in the core's private cache with appropriate access permissions, then it is a cache hit, and the core completes the memory request. If A is not present in the core's private cache with the appropriate access permissions, the cache controller generates a coherence message based on the memory request. The cache controller communicates the coherence message on the shared snooping bus in the core's allocated slot. The core that generated the ReadReq/WriteReq observes the coherence message for its request as an OwnRead/OwnWrite event, and other cores observe the same coherence message as OtherRead/OtherWrite events. Cores change the coherence state of their cached copies of A based on the observed event and the current coherence state of A in their private caches. Hence, the state transitions in a cache coherence protocol *map* the current coherence state and observed event to a new coherence state, and defined as the function $\mathbb{T} : \mathbb{S} \times \mathbb{E} \rightarrow \mathbb{S}$.

Multi-core coherence views

The latency analysis in Section 4.7 requires information of the coherence states of the cache line across *all* cores in the multi-core platform. This is because, the latency of a core's memory request to a cache line is dependent on the coherence states of the cache line in other cores, and the state changes due to memory activity on the cache line [74]. However, the above modeling of coherence states and transitions between coherence states is restricted to a *single* core. For example, consider the state transition from **I** to **S** on a write request in the PMSI protocol. The latency to complete this transition is dependent on whether or not another core has the same cache line in **M** state, and such information is unavailable from the **I** to **S** state transition. To this end, we present *multi-core coherence views* that capture the coherence states of a cache line across *all* cores in the multi-core platform, and changes to the coherence states of the cache line across all cores due to memory activity on the cache line. There are three main components of a multi-core coherence view: (1) state view (Definition 2), (2) event view (Definition 3), and (3) transitioned state view (Definition 4).

Definition 2. *A **state view** of* A*,* $sv_{\mathsf{A}} = \langle s_{\mathsf{A}}^{c_0}, s_{\mathsf{A}}^{c_1}, ..., s_{\mathsf{A}}^{c_{N-1}} \rangle$*, is an* N*-tuple composed of the coherence states of* A *across all* N *cores in a multi-core platform where* $\forall i \in [0, N-1]$*,* $c_i \in \mathbb{C}$ *and* $s_{\mathsf{A}}^{c_i} \in \mathbb{S}$*.*

Definition 3. *An **event view** of* A*,* $ev_{\mathsf{A}}^{c_i} = \langle e_{\mathsf{A}}^{c_0}, e_{\mathsf{A}}^{c_1}, ..., e_{\mathsf{A}}^{c_{N-1}} \rangle$*, is an* N*-tuple composed of events observed by all* N *cores in a multi-core platform due to* c_i*'s memory request to* A *where* $\forall i \in$

$[0, N-1]$, $c_i \in \mathbb{C}$, $e^{c_i}_{\mathsf{A}} \in \mathbb{E}$ *and* $e^{c_i}_{\mathsf{A}} = h(c_i, c_j, op)$ *as defined below:*

$$h(c_i, c_j, op) = \begin{cases} \textit{OwnRead} & : \textit{if } c_j = c_i \wedge op = \textit{ReadReq} \\ \textit{OtherRead} & : \textit{if } c_j \neq c_i \wedge op = \textit{ReadReq} \\ \textit{OwnWrite} & : \textit{if } c_j = c_i \wedge op = \textit{WriteReq} \\ \textit{OtherWrite} & : \textit{if } c_j \neq c_i \wedge op = \textit{WriteReq} \end{cases} \tag{4.1}$$

$op \in \mathbb{O}$ *is the memory request.*

Definition 4. *Let* $ev^{c_i}_{\mathsf{A}} = \langle e^{c_0}_{\mathsf{A}}, e^{c_1}_{\mathsf{A}}, ..., e^{c_{N-1}}_{\mathsf{A}} \rangle$ *be an event view caused by* c_i*'s memory request to* A*, and* $sv_{\mathsf{A}} = \langle s^{c_0}_{\mathsf{A}}, s^{c_1}_{\mathsf{A}}, ..., s^{c_{N-1}}_{\mathsf{A}} \rangle$ *be the state view of cache line* A *prior to* c_i*'s request to* A*. A* ***transitioned state view*** *of* A*,* $tv_{\mathsf{A}} = \langle s\prime^{c_0}_{\mathsf{A}}, s\prime^{c_1}_{\mathsf{A}}, ..., s\prime^{c_{N-1}}_{\mathsf{A}} \rangle$*, where* $s\prime^{c_i}_{\mathsf{A}} = \mathbb{T}(s^{c_i}_{\mathsf{A}}, e^{c_i}_{\mathsf{A}})$, $\forall c_i \in \mathbb{C}$ *is the transitioned coherence state obtained from applying the transition function* $\mathbb{T}$ *on event* $e^{c_i}_{\mathsf{A}}$ *and state* $s^{c_i}_{\mathsf{A}}$*.*

The transitioned state view describes the final coherence states of a cache line across all cores due to a core's memory request on the cache line. This view captures the state changes of the core that made the memory request and the state changes of other cores that observed the remote memory request. Note that the state views are specific to a cache coherence protocol as their compositions are dependent on the coherence states and transitions defined in the protocol state machine. For the rest of the paper, we use the shorthand representation $(sv_{\mathsf{A}}, ev^{c_i}_{\mathsf{A}}) \rightsquigarrow tv_{\mathsf{A}}$ to denote the following: a memory request to A by $c_i \in \mathbb{C}$, which results in an event view $e^{c_i}_{\mathsf{A}}$, on state view sv_{A} transitions to a new transitioned state view tv_{A}.

Illustrative example using PMSI protocol. The three states of the PMSI protocol are: (1) Modified (**M**), (2) Shared (**S**), and (3) Invalid (**I**). A cache line in **I** state denotes unavailability of data. Hence, the **I** state has $invalid$ access permissions, $clean$ data state, and $passive$ data authority. A core that has a cache line in **S** state has read only access permissions, and the core has not modified the cache line contents. The **S** state in PMSI has $read$ access permissions, $clean$ data state, and $passive$ data authority. A core that has a cache line in **M** state has read and write access permissions, and the core has modified the cache line contents. As a result, a core that has a cache line in **M** state must respond to a remote request to the same cache line with the updated data contents. The **M** state in PMSI has $write$ access permissions, $dirty$ data state, and $active$ data authority.

Consider a scenario where c_0 has A in the **M** state, c_1 does not have A (**I** state), and c_1 issues a write request to A. Based on the transitions in PMSI protocol, c_0 transitions from **M** to **I** state on observing a remote write request, and c_1 changes the coherence state of A from **I** to **M** on receiving the requested data. Since **M** has $active$ data authority, c_0 sends the requested data to c_1. For this scenario, $(sv_{\mathsf{A}}, ev^{c_1}_{\mathsf{A}}) \rightsquigarrow tv_{\mathsf{A}}$, $sv_{\mathsf{A}} = \langle \mathbf{M}, \mathbf{I} \rangle$, $ev^{c_1}_{\mathsf{A}} = \langle \mathsf{OtherWrite}, \mathsf{OwnWrite} \rangle$, and $tv_{\mathsf{A}} = \langle \mathbf{I}, \mathbf{M} \rangle$. Consider an alternate scenario where c_1 has A in **M** state, c_0 does not have A (**I** state), and c_0 issues a read request to A. Based on the transitions in the PMSI protocol, c_1

Table 4.1: 2-core $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_{ua}}) \rightsquigarrow tv_{\mathsf{A}}$ for PMSI protocol.

sv_{A}	ev_{A}	tv_{A}
$\langle \mathtt{I}, \mathtt{I} \rangle$	$\langle$OwnRead, OtherRead$\rangle$	$\langle \mathtt{S}, \mathtt{I} \rangle$
$\langle \mathtt{I}, \mathtt{I} \rangle$	$\langle$OwnWrite, OtherWrite$\rangle$	$\langle \mathtt{M}, \mathtt{I} \rangle$
$\langle \mathtt{S}, \mathtt{S} \rangle$	$\langle$OtherWrite, OwnWrite$\rangle$	$\langle \mathtt{I}, \mathtt{M} \rangle$
$\langle \mathtt{S}, \mathtt{S} \rangle$	$\langle$OwnRead, OtherRead$\rangle$	$\langle \mathtt{S}, \mathtt{S} \rangle$
$\langle \mathtt{M}, \mathtt{I} \rangle$	$\langle$OtherWrite, OwnWrite$\rangle$	$\langle \mathtt{I}, \mathtt{M} \rangle$
$\langle \mathtt{M}, \mathtt{I} \rangle$	$\langle$OwnWrite, OtherWrite$\rangle$	$\langle \mathtt{M}, \mathtt{I} \rangle$
$\langle \mathtt{I}, \mathtt{S} \rangle$	$\langle$OwnRead, OtherRead$\rangle$	$\langle \mathtt{S}, \mathtt{S} \rangle$
$\langle \mathtt{I}, \mathtt{S} \rangle$	$\langle$OtherRead, OwnRead$\rangle$	$\langle \mathtt{I}, \mathtt{S} \rangle$

transitions from **M** to **S** state on observing a remote read request, and c_0 changes the coherence state of A from **I** to **S** on receiving the requested data. For this scenario, $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_1}) \rightsquigarrow tv_{\mathsf{A}}$, $sv_{\mathsf{A}} = \langle \mathtt{I}, \mathtt{M} \rangle$, $ev_{\mathsf{A}}^{c_1} = \langle$OwnRead, OtherRead$\rangle$, and $tv_{\mathsf{A}} = \langle \mathtt{S}, \mathtt{S} \rangle$. Table 4.1 enumerates all possible unique $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_{ua}}) \rightsquigarrow tv_{\mathsf{A}}$ for the PMSI protocol for a 2-core system.

Operations on state views

We define operations that transform the state views in $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_i}) \rightsquigarrow tv_{\mathsf{A}}$ to numeric values. In Section 4.7, we use these operations to derive conditional expressions that identify the protocol state machine properties that impact the resulting WCAL bound of a memory request under the protocol. Definitions 5 and 6 define value functions for different components of the coherence state, Definition 7 defines a general weight operation for a state view, and Definition 8 captures the change in coherence states between an initial state view and a transitioned state view.

Definition 5. *$DSV(s_{\mathsf{A}}^{c_i})$, is the data state value of a coherence state, where $DSV : \mathbb{S} \rightarrow \{0, 1\}$ is defined as*

$$DSV(s_{\mathsf{A}}^{c_i}) = \begin{cases} 1 & : ds_{\mathsf{A}}^{c_i} = \textit{dirty} \\ 0 & : ds_{\mathsf{A}}^{c_i} = \textit{clean} \end{cases} \tag{4.2}$$

Definition 6. *$DAV(s_{\mathsf{A}}^{c_i})$, is the data authority value of a coherence state, where $DAV : \mathbb{S} \rightarrow \{0, 1\}$ is defined as*

$$DAV(s_{\mathsf{A}}^{c_i}) = \begin{cases} 1 & : da_{\mathsf{A}}^{c_i} = \textit{active} \\ 0 & : da_{\mathsf{A}}^{c_i} = \textit{passive} \end{cases} \tag{4.3}$$

Definition 7. *The f-weight of a state view sv_{A}, $\Sigma(sv_{\mathsf{A}}, f)$, is the sum of f-value of coherence states in sv_{A}.*

$$\Sigma(sv_{\mathsf{A}}, f) = \Sigma_{i=0}^{N-1} f(s_{\mathsf{A}}^{c_i}) \tag{4.4}$$

f can be one of the two value functions defined in Definitions 5-6. Hence, the data state weight and data authority weight of sv_{A} are $\Sigma(sv_{\mathsf{A}}, DSV)$, and $\Sigma(sv_{\mathsf{A}}, DAV)$, respectively.

Definition 8. *$\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, f)$ is the change in f-weight of state views for A when $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_i}) \rightsquigarrow tv_{\mathsf{A}}$.*

$$\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, f) = \Sigma(tv_{\mathsf{A}}, f) - \Sigma(sv_{\mathsf{A}}, f) \tag{4.5}$$

$\Sigma(tv_{\mathsf{A}}, f)$ and $\Sigma(sv_{\mathsf{A}}, f)$ are state view f-weights (Definition 7) of tv_{A} and sv_{A} respectively, and f can be one of the two value functions defined in Definitions 5 and 6. Hence, the change in data state weight and change in data authority weight for cache line A are:

$$\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) = \Sigma(tv_{\mathsf{A}}, DSV) - \Sigma(sv_{\mathsf{A}}, DSV) \tag{4.6}$$

$$\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) = \Sigma(tv_{\mathsf{A}}, DAV) - \Sigma(sv_{\mathsf{A}}, DAV) \tag{4.7}$$

4.6.2 Design principles of cache coherence protocols

Write-backs to shared memory and coherence message broadcasts are examples of *shared bus actions* executed by a core. A cache coherence protocol that causes a core to execute *nonessential* shared bus actions unnecessarily increases the latency of a core's memory request and complicates the cache coherence protocol design. We define four design principles that eliminate such nonessential shared bus actions in predictable cache coherence protocols. We use $\mathbb{P}$ to denote the set of predictable cache coherence protocols that follow these four design principles. The analysis in the following section holds for all predictable cache coherence protocols in $\mathbb{P}$. The PMSI and PMESI protocols described in Chapter 3, Section 3.6 are members of $\mathbb{P}$.

Property 1. *A core does not execute shared bus actions for a cache line if the data state and data authority of the cache line in the core's private cache do not change on own or other cores' memory requests to the cache line.*

A core's memory request to a cache line that is a cache hit does not change the coherence state of the cache line. This is because the cache line in the core's private cache has the appropriate coherence state (access permissions, data state, and data authority) to complete the memory request. Hence, the core does not require communication with other cores or the shared memory to complete the memory request, and as a result, shared bus actions are unnecessary. For example, in the PMSI and PMESI protocols, $(\mathtt{S}, \mathsf{OwnRead}/\mathsf{OtherRead}) \rightarrow \mathtt{S}$ does not generate shared bus actions.

Property 2. *Shared bus actions are not executed by a core on observing another core's memory request to a cache line if the data state and authority weights of the cache line across all cores before and after the memory request are equal.*

Equal data state and authority weights of a cache line before and after a memory request to the cache line means that the overall data authority and data state of the cache line across all cores remain *unchanged*. As a result, a core's shared bus actions in response to another core's memory request on a cache line in this scenario do *not* convey any new information about the data state and data authority of the cache line and hence, *unnecessary*. In the PMSI and PMESI protocols, the transition $(\texttt{M}, \mathsf{OtherWrite}) \rightarrow \texttt{I}$ does not generate shared bus actions as the core performing OwnWrite exercises the transition $(\texttt{I}, \mathsf{OwnWrite}) \rightarrow \texttt{M}$. Notice that before and after the memory request, only one core has the cache line in **M** state. Hence, shared bus actions by a core other than the requesting core are unnecessary as the cache line remains in the **M** state across all cores before and after the memory request.

Property 3. *A core does not change the data state or data authority of a cache line in its private cache from $dirty$ to $clean$ and $active$ to $passive$ respectively on its own memory request to the cache line.*

The rationale behind this property is that a core that has a cache line in $dirty$ data state or $active$ data authority means that the core has a *valid* version of the cache line in its private cache. This means that a core can complete its own memory request to this cache line with no changes to the data state or data authority. Therefore, shared bus actions associated due to changing data state from $dirty$ to $clean$ or data authority from $active$ to $passive$ are eliminated. In the PMSI and PMESI protocols, the following transitions are disallowed: $(\texttt{M}, \mathsf{OwnRead}) \rightarrow \texttt{S}$, $(\texttt{E}, \mathsf{OwnRead}) \rightarrow \texttt{S}$.

Property 4. *A core does not change the data authority of a cache line in its private cache from $passive$ to $active$ on observing another core's request to the same cache line.*

Recall from Section 4.6.1 that multiple cores can have a cache line with $passive$ data authority. Allowing cores to change their data authority from $passive$ to $active$ would result in multiple cores executing shared bus actions to change their data state and data authority. These need to be resolved such that only one core has the cache line with $active$ data authority, which unnecessarily complicates the protocol design. Hence, such changes to the data authority are disallowed, which eliminates the associated shared bus actions.

4.7 Worst-case Asymptotic Latency Analysis (WCAL)

In this work, we perform an *asymptotic analysis* that captures the growth behavior of memory request latency under a predictable cache coherence protocol with the number of cores (N). The function for the asymptotic analysis is a core's memory request latency under a predictable cache

coherence protocol. The asymptotic latency analysis abstracts away certain detailed features required to perform a precise latency analysis such as details regarding the arbitration schedule.

The *best-case asymptotic latency* of a core's memory request under a predictable cache coherence protocol is when it is a *cache hit*. Since a cache hit does not generate any coherence messages and does not require shared bus accesses, the best-case asymptotic latency is *independent* of the number of cores. On the other hand, the *worst-case asymptotic latency* (WCAL) of a core's memory request under a predictable cache coherence protocol is when the request is not available in the core's private cache (cache miss) and this request incurs timing interference due to simultaneous memory activity from other cores. *Bridging* the WCL gap requires understanding how the WCL of memory request grows with the number of cores and the design features of predictable cache coherence protocols that cause such growth. The WCAL analysis presented in this section concisely captures this understanding. The following lemmas and proofs derive the WCAL of a core's memory request under a predictable cache coherence protocol. We use the Big-O notation ($\mathcal{O}$) to denote the WCAL of a core's memory request.

Main result and proof overview. Our main result (Theorem 3) exposes the necessary and sufficient design properties of $p \in \mathbb{P}$ that render the WCAL of a memory request under p to grow quadratically with the number of cores ($\mathcal{O}(N^2)$). $c_{ua} \in \mathbb{C}$ denotes the core under analysis. Lemma 8 establishes the design properties of p that cause a core $c_i \in \mathbb{C} \setminus \{c_{ua}\}$ to execute shared bus actions in response to c_{ua}'s memory request. Lemma 9 uses the result of Lemma 8 to prove that c_i's shared bus responses must complete before c_{ua} can receive its requested data. Theorem 3 proves that the WCAL of memory request under $p \in \mathbb{P}$ that satisfies Lemma 9 is $\mathcal{O}(N^2)$.

Definition 9. *When* $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_{ua}}) \rightsquigarrow tv_{\mathsf{A}}$, $\alpha(sv_{\mathsf{A}}, tv_{\mathsf{A}})$ *is a boolean expression computed as*

$$\begin{aligned}&\Big((\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) < 0) \wedge (DSV(sf_{\mathsf{A}}^{c_{ua}}) = DSV(s_{\mathsf{A}}^{c_{ua}}))\Big) \vee \\ &\Big((\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) < 0) \wedge (DAV(sf_{\mathsf{A}}^{c_{ua}}) = DAV(s_{\mathsf{A}}^{c_{ua}}))\Big)\end{aligned} \tag{4.8}$$

Lemma 8. *When* $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_{ua}}) \rightsquigarrow tv_{\mathsf{A}}$, $\exists\ c_i \in \mathbb{C} \setminus \{c_{ua}\}$ *that executes shared bus actions in response to* c_{ua}*'s memory request to* $\mathsf{A} \iff \alpha(sv_{\mathsf{A}}, tv_{\mathsf{A}})$ *evaluates to true.*

Direct Proof. **Proof for $\Longrightarrow$.** When $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_{ua}}) \rightsquigarrow tv_{\mathsf{A}}$, let $\exists\ c_i \in \mathbb{C} \setminus \{c_{ua}\}$ that executes shared bus actions in response to c_{ua}'s memory request to A. Since c_{ua} and c_i execute shared bus actions, the data state and data authority of A in c_{ua} and c_i changes (Property 1). We consider four cases based on the changes in data state and data authority weights (Definition 8) when $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_{ua}}) \rightsquigarrow tv_{\mathsf{A}}$. We prove that c_i executes shared bus actions in cases 1, 2 and 3. For case 4, we arrive at a contradiction based on the properties defined in Section 4.6.2, and therefore, c_i does not execute shared bus actions.

1. $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) < 0$

2. $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) < 0$

3. $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) < 0 \wedge \Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) < 0$

4. $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) \geq 0 \wedge \Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) \geq 0$

Case 1. In this case, $\Sigma(sv_{\mathsf{A}}, DSV) > 0$ before c_{ua}'s memory request and $\Sigma(sv_{\mathsf{A}}, DSV) = 0$ after c_{ua}'s memory request. As a result, $\exists c_j \in \mathbb{C}$ that had A with $dirty$ data state before c_{ua}'s memory request and changed its data state to $clean$ after c_{ua}'s memory request. Since Property 3 disallows a core to change its data state from $dirty$ to $clean$ on its own memory request, $c_j \neq c_{ua}$. Hence, $c_j = c_i$. Furthermore, since multiple cores cannot have a cache line with $dirty$ data state at the same time, the data state of c_{ua} must be $clean$ before and after its memory operation. Hence, c_i executes shared bus actions in response to c_{ua}'s memory request to A when $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) < 0) \wedge (DSV(s\prime_{\mathsf{A}}^{c_{ua}}) = DSV(s_{\mathsf{A}}^{c_{ua}}))$ evaluates to true.
Case 2. Following the approach of case 1 and applying Propery 3, c_i changes the data authority of A in its private cache from $active$ to $passive$ on observing c_{ua}'s memory request. Hence, c_i executes shared bus actions in response to c_{ua}'s memory request to A when $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) < 0) \wedge (DAV(s\prime_{\mathsf{A}}^{c_{ua}}) = DAV(s_{\mathsf{A}}^{c_{ua}}))$ evaluates to true.
Case 3. This case combines cases 1 and 2; hence, c_i executes shared bus actions in response to c_{ua}'s memory request.
Case 4. There are four sub-cases:
Case 4.1: $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) = 0 \wedge \Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) = 0$ Under an efficient predictable cache coherence protocol, Property 2 states that shared bus actions are unnecessary when $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) = 0$ and $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) = 0$. Hence, we reach a contradiction, and therefore, c_i does not execute shared bus actions in this case.
Case 4.2: $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) > 0 \wedge \Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) > 0$ When $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) > 0$, $\Sigma(sv_{\mathsf{A}}, DSV) = 0$ before the memory request and $\Sigma(tv_{\mathsf{A}}, DSV) > 0$ after the memory request. This means that the data state of A across all cores and shared memory was $clean$ before c_{ua}'s memory request, and $\exists\ c_j \in \mathbb{C}$ updated A and changed its data state of A from $clean$ to $dirty$. Now, $c_j \neq c_i$ as a core cannot change its data state to $dirty$ on observing another core's request; a core changes the data state of a cache line to $dirty$ due to data content modifications caused by an own memory operation. Hence, $c_j = c_{ua}$. This means that the other cores ($\mathbb{C} \setminus \{c_{ua}\}$), which includes c_i, do not change their data states of A.

Similarly, other cores in $\mathbb{C} \setminus \{c_{ua}\}$, which includes c_i, do not change their data authority of A when $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) > 0$; Property 4 disallows a core to change their data authority from $passive$ to $active$ on observing a remote memory request. Notice that in both cases ($\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) > 0$ and $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) > 0$), c_i does not change data authority and data state of A in its private cache. Hence, from Property 1, c_i does not execute shared bus actions when $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) > 0$ and $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) > 0$.

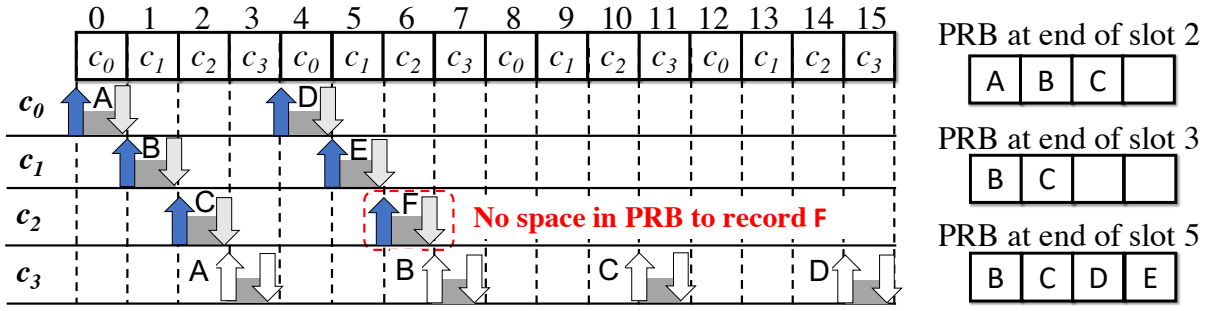


Figure 4.4: Visual aid for Lemma 9.

Case 4.3: $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) = 0 \wedge \Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) > 0$ and
Case 4.4: $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) = 0 \wedge \Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) > 0$ The proofs for these cases follows from cases 4.1 and 4.2, and hence, c_i cannot execute shared bus actions.

Proof for $\Longleftarrow$. When $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_{ua}}) \rightsquigarrow tv_{\mathsf{A}}$, let $\alpha(sv_{\mathsf{A}}, tv_{\mathsf{A}})$ evaluate to true. For $(\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) < 0) \wedge (DSV(s\prime_{\mathsf{A}}^{c_{ua}}) = DSV(s_{\mathsf{A}}^{c_{ua}}))$ to be true, c_{ua}'s data state must not change when $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_{ua}}) \rightsquigarrow tv_{\mathsf{A}}$, and $\Sigma(sv_{\mathsf{A}}, DSV) > 0$, $\Sigma(tv_{\mathsf{A}}, DSV) = 0$. The scenario that satisfies this is when $\exists\, c_i \in \mathbb{C} \setminus \{c_{ua}\}$ that has A in *dirty* data state in sv_{A}, and the memory request from c_{ua} causes all cores to have A in *clean* data state. Hence, c_i must write-back the data of A to shared memory in response to c_{ua}'s memory request, which is a shared bus action.

For $(\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) < 0) \wedge (DAV(s\prime_{\mathsf{A}}^{c_{ua}}) = DAV(s_{\mathsf{A}}^{c_{ua}}))$ to be true, c_{ua}'s data authority must not change when $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_{ua}}) \rightsquigarrow tv_{\mathsf{A}}$, and $\Sigma(sv_{\mathsf{A}}, DAV) > 0$ and $\Sigma(sv_{\mathsf{A}}, DAV) = 0$. The scenario that satisfies this is when $\exists\, c_i \in \mathbb{C} \setminus \{c_{ua}\}$ that has A in *active* data authority in sv_{A}, and c_{ua}'s memory request to A causes c_i to change its data authority to *passive*. When c_i changes its data authority of A from *active* to *passive*, the shared memory is the data source of A. Hence, c_i must communicate coherence messages to inform the shared memory of this change in data authority. □

The key takeaway from Lemma 8 is that a negative change to the overall data state or data authority of a cache line across all cores on a core's (c_{ua}) memory request causes another core ($c_i \in \mathbb{C} \setminus \{c_{ua}\}$) to respond with shared bus accesses.

Lemma 9. *When $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_{ua}}) \rightsquigarrow tv_{\mathsf{A}}$, if $\alpha(sv_{\mathsf{A}}, tv_{\mathsf{A}})$ evaluates to true, then c_{ua} must wait for shared bus actions by $\forall c_i \in \mathbb{C} \setminus \{c_{ua}\}$, which are in response to its memory request, to complete before it can receive* A.

Proof by Contradiction. We show that there exists a scenario where allowing c_i to complete shared bus actions after c_{ua}'s memory request results in c_i's pending response buffer to overflow

(record more than N responses). We consider two cases and use Figure 4.4 as a visual aid for the first case.

Case 1: $N > 2$. Let c_{ua} receive the most up-to-date data contents of A in the same time slot when it made the request. c_i performs shared bus actions due to changes in either its data state or data authority of A in response to c_{ua}'s memory request. These shared bus actions are inserted into c_i's pending response buffer. In Figure 4.4, c_{ua} is c_0 and c_i is c_3. c_3 has modified versions of A-F in its private cache. Hence, c_3 inserts the write-back response for A in its pending response buffer (PRB) on observing c_0's read request to A. c_i cannot execute these shared bus actions in the current time slot, which is allocated to c_{ua}, and must wait for its next allocated slots to complete these actions. This is shown in Figure 4.4 where c_3 executes the write-back for A in ❸. For an arbitration scheme that allocates at least one slot to each core, the worst-case arbitration latency is N time slots. This means that the worst-case arbitration latency scales *linearly* with core count $\mathcal{O}(N)$. In the worst-case, the remaining cores ($C \setminus \{c_i\}$) can make requests to different cache lines other than A that c_i has modified. This means that c_i's pending response buffer has $N-1$ responses at the start of its allocated slot. In Figure 4.4, c_3's PRB has three write-back responses in its PRB at the start of ❸ (write-back responses to A, B, and C). Since other cores received their requested data in the same slot when they made their requests, these cores can make new requests in the next arbitration period. In the worst case, these cores make memory requests to different cache lines also modified by c_i. As a result, c_i has to record $2 \times (N-1) > N$ pending responses before its next allocated slot. c_i cannot record more than N write-back responses (due to pending response buffer capacity), and hence, this scenario cannot happen. In Figure 4.4, c_0-c_2 make new requests to D, E, and F in ❹, ❺, and ❻ that are modified by c_3, and hence, c_3 must queue up the write-back responses to these cache lines in its PRB. However, c_3's PRB is full on inserting the write-back for E and does not have enough entries for F.

Case 2: $N = 2$. For this case, consider an arbitration scheme that allocates multiple time slots to one core (c_{ua}) and one time slot to another core (c_i) such as a weighted TDM arbitration scheme. Since c_i can finish one write-back to shared memory in an arbitration period, this case also renders the same situation as described in the previous case where c_i's pending response buffer can overflow. □

The key takeaway from Lemma 9 is that a core (c_{ua}) must wait for other cores ($\mathbb{C} \setminus \{c_{ua}\}$) to complete their shared bus accesses that are in response to its memory request before it can receive the requested data and complete its request.

Theorem 3. *For a predictable cache coherence protocol $p \in \mathbb{P}$, if $\exists\, sv_{\mathsf{A}}, tv_{\mathsf{A}}$ such that $\alpha(sv_{\mathsf{A}}, tv_{\mathsf{A}})$ evaluates to true when $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_{ua}}) \rightsquigarrow tv_{\mathsf{A}}$, then the WCAL of a memory request under p is $\mathcal{O}(N^2)$.*

Direct proof. Our proof consists of two parts. The first part establishes that in the worst-case, $N-1$ cores can have pending memory requests when c_{ua} communicates its memory request to

A. The second part establishes that c_{ua} must wait for the prior cores to complete their requests before c_{ua} can receive data for A and complete its memory request.

From Section 4.5, the predictable arbitration policy deployed on the shared message bus allocates at least one time slot to each core. In the worst-case, cores in $\mathbb{C} \setminus \{c_{ua}\}$ have allocated slots before c_{ua}'s allocated slot. Hence, $|\mathbb{C} \setminus \{c_{ua}\}| = N - 1$ cores can communicate memory requests in their allocated slots before c_{ua} communicates its memory request to A. In the worst-case, $\forall c_i \in \mathbb{C} \setminus \{c_{ua}\}$, $(sv_{\mathsf{X}_i}, ev_{\mathsf{X}_i}^{c_i}) \rightsquigarrow tv_{\mathsf{X}_i}$ satisfies $\alpha(sv_{\mathsf{X}_i}, tv_{\mathsf{X}_i})$ (Lemma 8) where X_i is the cache line requested by c_i. As a result, for each $c_i \in \mathbb{C} \setminus \{c_{ua}\}$, $\exists\ c_j \in \mathbb{C}$ where $c_i \neq c_j$ that executes shared bus actions in response to c_i's memory request. From Lemma 9, c_i must wait for c_j's bus actions before it can complete its memory request. Hence, $\forall c_i \in \mathbb{C} \setminus \{c_{ua}\}$ cannot receive their data and complete their memory requests in the same slots that they communicated their requests in. As a result, $N - 1$ cores can have pending requests when c_{ua} communicates its memory request to A.

A core's pending response buffer records at most N responses to distinct cache lines, and processes the responses in the response buffer in FIFO order (Section 4.5). In the worst-case, core c_i's pending response buffer of $c_i \in \mathbb{C} \setminus \{c_{ua}\}$ has $N - 1$ responses where the last response is for c_{ua}'s memory request. This scenario can happen when $\forall c_i \in \mathbb{C} \setminus \{c_{ua}\}$, $\mathsf{X}_i \neq \mathsf{A}$ and $\forall c_i, c_j \in \mathbb{C} \setminus \{c_{ua}\}$ and $c_i \neq c_j$, $\mathsf{X}_i \neq \mathsf{X}_j$. As a result, c_i responds to c_{ua}'s memory request after executing $N - 1$ shared bus actions for the prior $N - 1$ memory requests.

Recall that the shared bus arbitration policy allocates each core a constant number of time slots (Section 4.5), and the arbitration scales linearly with the number of cores $\mathcal{O}(N)$. Since c_i takes $\mathcal{O}(N)$ to execute a shared bus action, c_{ua} receives A after c_i completes $N - 1$ bus actions, which results in WCAL of a memory request to be $(N - 1) \times \mathcal{O}(N) = \mathcal{O}(N^2)$. □

The key takeaway from Theorem 3 is that the WCL of a memory request under a predictable cache coherence protocol that exhibits at least one $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_{ua}}) \rightsquigarrow tv_{\mathsf{A}}$ *that satisfies* $\alpha(sv_{\mathsf{A}}, tv_{\mathsf{A}})$ *grows quadratically with the number of cores* N.

4.7.1 Applying the formal model and analysis

We apply the presented formal model to derive the WCAL of a memory request under the PMESI protocol described in Chapter 3 (Section 3.6). For a 2-core multi-core configuration, the set of valid state views constructed based on the PMESI protocol is $\{\langle \mathbf{I}, \mathbf{I}\rangle, \langle \mathbf{I}, \mathbf{E}\rangle, \langle \mathbf{I}, \mathbf{M}\rangle, \langle \mathbf{I}, \mathbf{S}\rangle, \langle \mathbf{E}, \mathbf{I}\rangle, \langle \mathbf{M}, \mathbf{I}\rangle, \langle \mathbf{S}, \mathbf{I}\rangle, \langle \mathbf{S}, \mathbf{S}\rangle\}$. The following $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{c_i}) \rightsquigarrow tv_{\mathsf{A}}$ in PMESI protocol satisfy $\alpha(sv_{\mathsf{A}}, tv_{\mathsf{A}})$: $(\langle \mathbf{E}, \mathbf{I}\rangle, \langle \mathsf{OtherRead}, \mathsf{OwnRead}\rangle) \rightsquigarrow \langle \mathbf{S}, \mathbf{S}\rangle$ and $(\langle \mathbf{M}, \mathbf{I}\rangle, \langle \mathsf{OtherRead}, \mathsf{OwnRead}\rangle) \rightsquigarrow \langle \mathbf{S}, \mathbf{S}\rangle$. Consider $(\langle \mathbf{E}, \mathbf{I}\rangle, \langle \mathsf{OtherRead}, \mathsf{OwnRead}\rangle) \rightsquigarrow \langle \mathbf{S}, \mathbf{S}\rangle$. Since $\mathbf{E}$ has *dirty* data state and $\mathbf{I}$ has *clean* data state, $\Sigma(sv_{\mathsf{A}}, DSV) = 1{+}0 = 1$. $\Sigma(tv_{\mathsf{A}}, DSV) = 0{+}0 = 0$ as $\mathbf{S}$ has *clean* data state. Hence, $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) = 0 - 1 = -1$, which is < 0. Furthermore, since the core generating

the OwnRead moves from **I** to **S** state, $DSV(sI_{\mathsf{A}}^{cua}) = DSV(s_{\mathsf{A}}^{cua})$. Since there exists *at least one* sv_{A} and tv_{A} in PMESI protocol that satisfies $\alpha(sv_{\mathsf{A}}, tv_{\mathsf{A}})$, from Theorem 3, the WCAL of a memory request under PMESI grows quadratically with N ($\mathcal{O}(N^2)$).

In the following section, we describe one technique of tightening the WCL of a memory request under a predictable cache coherence protocol. This technique uses protocol changes to eliminate all $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{cua}) \rightsquigarrow tv_{\mathsf{A}}$ in the cache coherence protocol that satisfy $\alpha(sv_{\mathsf{A}}, tv_{\mathsf{A}})$. These protocol changes target the *offending transitions*, which we define in Definition 10.

Definition 10. *For a* $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{cua}) \rightsquigarrow tv_{\mathsf{A}}$ *that satisfies* $\alpha(sv_{\mathsf{A}}, tv_{\mathsf{A}})$*, the state transitions constituting* $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{cua}) \rightsquigarrow tv_{\mathsf{A}}$ *are defined as offending transitions.*

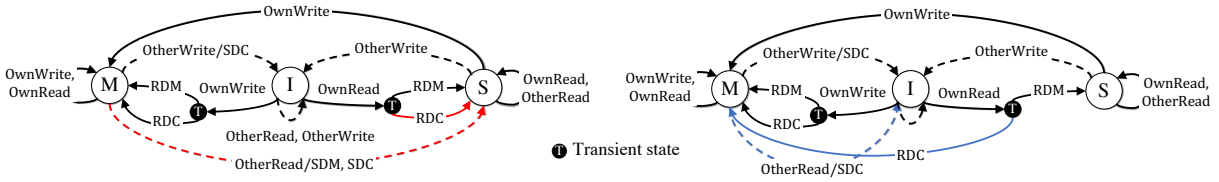
4.8 Tightening WCL bounds

In Section 4.4, we described protocol changes to the PMSI protocol to tighten its WCL, which resulted in the PMI protocol. These changes targeted the *offending transitions* (**M**, OtherRead) $\rightarrow$ **S** and (**I**, OwnRead) $\rightarrow$ **S** in the PMSI protocol. However, despite the improvements to WCAL under PMI compared to that under the PMSI protocol ($\mathcal{O}(N)$ vs $\mathcal{O}(N^2)$), the PMI protocol is an *inferior* data communication choice due to its significant performance slowdown. In order to design predictable cache coherence mechanisms that have tight WCAL ($\mathcal{O}(N)$ WCAL) while maintaining their average-case performance benefits, we need to *rethink* our identification of offending transitions. In the following paragraphs, we show that exposing events related to *data communication* on state transitions can *refine* the identification of offending transitions.

We revisit the PMSI protocol in Figure 4.5 to highlight our approach to identifying offending transitions using data communication events. The PMSI protocol in Figure 4.5a shows the following additional information related to *data communication* between cores and shared memory: (1) transient states (T) that denote waiting for data responses, and (2) events on transitions related to data communication. There are four types of events related to data communication: (1) send data to requesting core (SDC), (2) send data to shared memory (SDM) [1], (3) receive data from another core (RDC), and (4) receive data from shared memory (RDM). Note that this additional information regarding data communication events is *extracted* from our formal model. The data authority encoding in coherence states and the result of Lemma 8 identify the type of data communication events on transitions.

We make two observations from Figure 4.5a. First, in PMSI, a transition triggered on OwnRead or OwnWrite reaches the same destination state *irrespective* of the source of data. For example, a core that does not have a cache line and generates an OwnRead moves to **S** state irrespective of whether it receives the requested data from another core or from the shared mem-

[1] SDM is same as write-back to shared memory



(a) PMSI protocol with data communication events.

(b) PMSI* protocol with O(N) WCAL.

Figure 4.5: Transforming PMSI protocol to PMSI* protocol with $\mathcal{O}(N)$ WCAL. Transitions highlighted in red are offending transitions.

ory. Second, a core that has a cache line in **M** sends the data to the requesting core and performs a write-back to shared memory only when the cache line in the requesting core transitions from **I** to **S** on a OwnRead. In other words, (**I**, OwnRead) $\rightarrow$ **S** receives data from another core (RDC) only if there exists another core that has the data in **M** state.

These two observations allow us to *refine* the conditions under which (**I**, OwnRead) $\rightarrow$ **S** is an offending transition: (***I***, *OwnRead*) $\rightarrow$ ***S*** *is an offending transition when it receives the cache line from another core (RDC). Otherwise,* (***I***, *OwnRead*) $\rightarrow$ ***S*** *is not an offending transition if it receives data from the shared memory.* The shared memory sends a cache line to the requesting core *if and only if* there does not exist another core that has the same cache line in **M** state. Hence, if a core receives the requested cache line from the shared memory, then no cores will perform shared bus accesses in response to the requesting core's request. As a result, the **S** state does not increase the WCAL under this scenario, and can be retained.

Figure 4.5b shows the protocol changes to PMSI with this refinement regarding (**I**, OwnRead) $\rightarrow$ **S** resulting in the PMSI* protocol. The transition (**M**, OtherRead) $\rightarrow$ **S** is changed to (**M**, OtherRead) $\rightarrow$ **I**, which is the same in PMI. However, only a portion of the (**I**, OwnRead) $\rightarrow$ **S** is changed depending on the data source. In contrast, the protocol changes in PMI replaced (**I**, OwnRead) $\rightarrow$ **S** with (**I**, OwnRead) $\rightarrow$ **M** irrespective of the data source. In Figure 4.5b, cache line in transient state (T), moves to **M** on receiving data from another core (RDC) or to **S** on receiving data from shared memory (RDM). As a result, the PMSI* retains the **S** state, which allows *some* scenarios where multiple cores can simultaneously have a cache line in their private caches. In PMSI*, cores cannot simultaneously have a cache line in their private caches when one core has a cache line in **M** state and observes a read request from another core. In Section 4.9, we show that this loss in caching scenarios in PMSI* in at most 22% performance performance slowdown. Theorem 4 proves that the WCAL of PMSI* is $\mathcal{O}(N)$.

Theorem 4. *The WCL of a memory request under PMSI* is the same as that under the cache bypassing mechanism.*

Direct proof. In the cache bypassing mechanism, the worst-case instance is when the requested cache line is in the shared memory. As a result, the requesting core must wait for its next allocated

Protocol	Offending transitions	Protocol changes
PMSI	(M, OtherRead) → S (I, OwnRead) → S	(M, OtherRead) → I (I, OwnRead) → M on RDC (I, OwnRead) → S on RDM
PMESI	(M, OtherRead) → S (E, OtherRead) → S (I, OwnRead) → S	(M, OtherRead) → I (E, OtherRead) → I (I, OwnRead) → E on RDC (I, OwnRead) → S on RDM

Table 4.2: Protocol changes to PMSI and PMESI protocols.

time slot, which in the worst-case is the arbitration period. We show that PMSI* has the same worst-case instance as the cache bypassing mechanism.

In the PMSI* protocol described in Figure 4.5b, a core receives the requested cache line either from the shared memory (RDM) or from another core (RDC). A core receives the requested cache line from the shared memory only when there does not exist another core that has the same cache line in M state. Hence, there does not exist any cores that will respond with shared bus accesses in response to the requesting core's memory request. Therefore, the worst-case instance when a core receives a cache line from the shared memory is the same as that of cache bypassing. In the worst-case, the requesting core waits for its next allocated slot to broadcast its request, and will receive the requested cache line in the same slot.

On the other hand, a core receives the requested cache line from another core that has the cache line in M state. However, the transitions in PMSI* causes the core that has the cache line in M state to transition to I state after sending the cache line data (SDC). Furthermore, a core that receives the requested cache line from another core moves to M state. The corresponding $(sv_{\mathsf{A}}, ev_{\mathsf{A}}^{cua}) \rightsquigarrow tv_{\mathsf{A}}$ in this scenario is $(\langle M, I, \rangle, \langle \mathsf{OtherRead}, \mathsf{OwnRead} \rangle) \rightsquigarrow \langle I, M \rangle$. Applying the state view operations, $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DAV) = 0$ and $\Delta(sv_{\mathsf{A}}, tv_{\mathsf{A}}, DSV) = 0$. Hence, $\alpha(sv_{\mathsf{A}}, tv_{\mathsf{A}})$ is false. Since no cores respond with shared bus accesses, the requesting core receives the cache line in the same slot it broadcasted its memory request. □

Table 4.2 tabulates the offending transitions in the PMSI and PMESI protocols and summarizes the corresponding changes to these transitions applied by our technique. Note that our technique only makes changes to the protocol state machine at the private cache level; the protocol state machine at the shared memory is unchanged. The changes to the PMESI protocol are similar to those described for the PMSI protocol. This highlights the generality of our technique across different protocols, which is a consequence of our systematic approach. Tables 4.3 and 4.4 tabulate the complete PMSI* and PMESI* protocol state machines at the private cache level. The protocol state machines at the shared memory remain the same as those described in Tables 3.3 and 3.4 in Chapter 3. Cells marked in red denote the protocol changes that tighten the WCL

as described in Table 4.2.

As described earlier, the protocol changes in PMSI* and PMESI* rely on point-to-point data interconnects that allow for direct data communication between cores. As a result, a core receives data either through the shared data bus or through the point-to-point data interconnects. We assume that the shared data bus and point-to-point data interconnects have different ports into a core's cache memory. This makes the identification of the data source straightforward. A core's cache controller, which implements the coherence protocol, checks the data ports to determine the transitions to exercise based on the data source (RDC, RDM).

	Core events			Bus events						
States	Read	Write	Replacement	OwnGetS	OwnGetM	OwnData	OwnPutM	OtherGetS	OtherGetM	OtherPutM
I	Issue GetS()/IS_D	Issue GetM()/IM_D						—	—	—
S	Hit, Complete read	Issue Upg()/SM_A	-/I					—	I	
M	Hit, Complete read	Hit, Complete write	Issue PutM()/MI_A					Send data/I	Send data/I	
IS_D						If shared memory is data source, then complete read/S, else complete read/M		—	-/IS_DI	—
IM_D						Complete write/M		-/IM_DI	-/IM_DI	—
SM_A	Hit, Complete read/-		Stall		Complete write/M				Reissue write/I	
MI_A	Hit, Complete read	Hit, Complete write					Write-back data/I	Send data/II_A	Send data/II_A	
IM_DI						Complete write, Send data/I		—	—	—
IS_DI						Complete read/I		—	—	—
II_A							-/I	—	—	—

Table 4.3: Private memory states for PMSI* protocol. *issue msg/state* means the core issues the message *msg* and move to state *state*. A core issues a *read/write* request. Once the cache line is available, the core *reads/writes* it. A replacement triggers a cache line eviction. Highlighted cells denote impossible scenarios, and cells marked with '—' denote no change in state.

	Core events			Bus events						
States	Read	Write	Replacement	OwnGetS	OwnGetM	OwnData	OwnPutM	OtherGetS	OtherGetM	OtherPutM
I	Issue GetS()/IS_D	Issue GetM()/IM_D						—	—	—
S	Hit, Complete read	Issue Upg()/SM_A	-/I					—	I	
M	Hit, Complete read	Hit, Complete write	Issue PutM()/MI_A					Send data/I	Send data/I	
E	Hit, Complete read	Complete write/M	Issue PutM()/EI_A					Send data/I	Send data/I	
IS_D						If shared memory is data source, then complete read/S, else complete read/E		—	-/IS_DI	—
IM_D						Complete write/M		-/IM_DI	-/IM_DI	—
SM_A	Hit, Complete read/-		Stall		Complete write/M				Reissue write/I	
MI_A	Hit, Complete read	Hit, Complete write					Write-back data/I	Send data/II_A	Send data/II_A	
EI_A	Hit, Complete read	Complete write/MI_A					Write-back data/I	Send data/II_A	Send data/II_A	
IM_DI						Complete write, Send data/I		—	—	—
IS_DI						Complete read/I		—	—	—
II_A							-/I	—	—	—

Table 4.4: Private memory states for PMESI* protocol. *issue msg/state* means the core issues the message *msg* and move to state *state*. A core issues a *read/write* request. Once the cache line is available, the core *reads/writes* it. A replacement triggers a cache line eviction. Highlighted cells denote impossible scenarios, and cells marked with '—' denote no change in state.

4.9 Evaluation

We prototype the original and transformed predictable cache coherence protocols on the gem5 micro-architectural simulator [17]. The original protocols are the PMSI, and PMESI protocols and the new coherence protocols derived on applying our technique are PMSI*, and PMESI*. Table 4.5 describes our simulated real-time multi-core platform. We configure the shared LLC such that all accesses to the LLC are cache hits. We do this in order to focus on the impact of maintaining cache coherence on the memory request latency. We also compare against the cache bypassing mechanism [25, 61, 86].

Our evaluation uses synthetic workloads and the multi-threaded workloads in the SPLASH-2 benchmark suite [157]. The synthetic workloads stress different data communication patterns between cores, and exercise the protocol transitions. The workloads in the SPLASH-2 benchmark suite are derived from domains such as scientific computation and graphics. We run all SPLASH-2 workloads until completion, and check for data correctness using the in-built verification routines. We verified the data correctness for all the predictable protocols evaluated in this work.

4.9.1 Observed WCL

Table 4.6 shows the observed total WCL and analytical total WCL bounds for the cache bypassing mechanism, and the predictable cache coherence protocols for synthetic workloads. Recall that the synthetic workloads feature intensive data communication between cores, and hence, frequently stress the worst-case scenarios. The observed WCL is the maximum memory request latency observed across all synthetic benchmarks for a core configuration. The analytical bounds are computed based on the core count (N) and the TDM slot width S. The analytical total WCL bound for cache bypassing, PMSI*, and PMESI* is computed as $N \times S + S$ cycles (Theorem 4). The analytical total WCL bound for PMSI, and PMESI protocols is computed as $2NS \times (N + 1) + S$ [74].

Observations. Table 4.6 shows the widening WCL gap with core count between cache bypassing and existing predictable cache coherence protocols PMSI, and PMESI with core count. For 8-core and 16-core, the WCL gap is 16× and 32× respectively. On the other hand, the PMSI*, and PMESI* protocols have the same analytical WCL bounds as cache bypassing for different core counts. Furthermore, the observed WCL under PMSI* and PMESI* protocol are within the analytical WCL bounds.

Parameter	Configuration
Multi-core platform	4-16 in-order cores, one outstanding memory request per core, 2GHz operating frequency
Cache hierarchy	Private L1 split data and instruction caches, 32kB 4-way cache associative, shared L2 cache (last level cache LLC), cache line size 64-bytes
Bus interconnect	Shared data and message bus interconnects between cores and LLC, TDM arbitration policy, point-to-point data interconnects between cores, one TDM slot per core, slot width = 50 cycles

Table 4.5: Simulation parameters.

Cores	Cache bypassing		PMSI		PMESI		PMSI*		PMESI*	
	Bound	Obs	Bound	Obs	Bound	Obs	Bound	Obs	Bound	Obs
4	250	250	2050	1599	2050	1019	250	250	250	250
8	450	450	7250	6384	7250	5964	450	450	450	450
16	850	850	27250	24770	27250	23999	850	850	850	850

Table 4.6: Observed WCL (Obs) and analytical WCL bounds (Bound) in cycles for 4-core, 8-core, and 16-core configurations.

4.9.2 Average-case performance

Figure 4.6 shows the average execution time speedup of the different predictable cache coherence protocols compared to the cache bypassing data communication mechanism across all SPLASH-2 workloads. For the cache bypassing mechanism, we modified the SPLASH-2 applications to mark memory regions communicated between cores, and disallowed caching of memory addresses in these memory regions. We normalize the average speedup across synthetic benchmarks to the cache bypassing mechanism. Figures 4.7-4.8 empirically highlight the trade-off in performance observed in PMSI* and PMESI* protocols due to the protocol changes. Figure 4.7 shows the performance slowdown of the PMSI* protocol compared to the PMSI protocol, and Figure 4.8 shows the performance slowdown of the PMESI* protocol compared to the PMESI protocol A performance slowdown greater than 1 means that the execution time under the new protocol (PMSI*, PMESI*) is slower than the original protocol (PMSI, PMESI). For this evaluation, we use a multi-core configuration with 8-cores. We also evaluated the performance on the synthetic workloads, and observed similar performance trends.

Observations. From Figure 4.6, all coherence protocols including the new protocols offer significant performance benefits (as high as 5× performance speedup, 65% average performance speedup) over cache bypassing for the SPLASH-2 benchmarks. This is because the coherence protocols does not place caching constraints on data. Even though the new protocols sacrifice

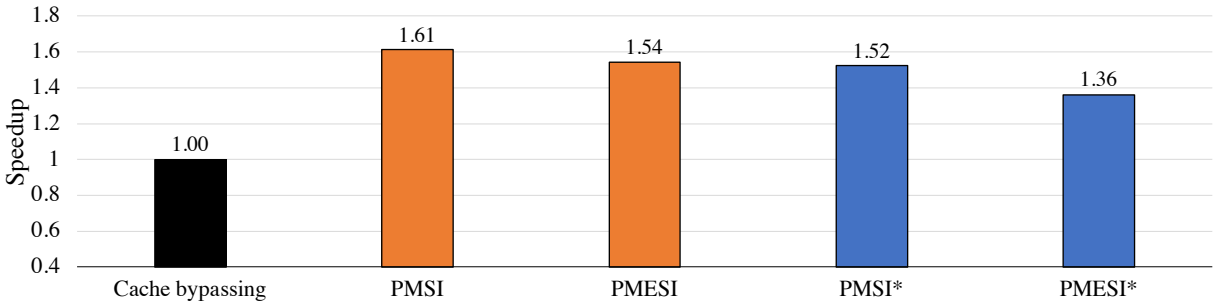


Figure 4.6: Average-case performance for SPLASH-2 workloads.

some performance opportunities compared to the original protocols, they still maintain between 36%-56% average performance speedup over cache bypassing while exhibiting the *same* WCL. We use Figures 4.7-4.8 to explain the performance trends between the transformed and original protocols for each SPLASH-2 benchmark.

From Figure 4.7, the PMSI* protocol trades-off minimal performance benefits (6% average slowdown) for tighter WCL bounds compared to the PMSI protocol. The PMSI* protocol disallows certain instances where multiple cores can have the same cache line in their private caches simultaneously. In particular, cores cannot simultaneously have a cache line in their private caches when one core has a cache line in **M** state and observes a read request from another core; the core with cache line **M** invalidates and moves to **I** and the cache line in the requesting core moves from **I** to **M**. Similarly, Figure 4.8 show that the PMESI* protocols trade-off minimal performance benefits (13% average slowdown respectively) for tighter WCL bounds compared to the PMESI protocol. Note that there are instances where PMSI* and PMESI* performance better (slowdown less than 1) compared to the PMSI and PMESI protocols respectively. For example, PMSI* and PMESI* exhibit 2%-3% performance improvement over the PMSI and PMESI protocols for the FFT benchmark. We attribute this to the reduced write-backs to shared memory in PMSI* and PMESI* protocols, which reduces the contention on a core's allocated time slots. Shared Memory write-backs and demand requests from a core contents for the core's allocated slots. Hence, we observe some instances where reducing the shared memory write-backs (FFT, Raytrace) improves performance. In summary, the technique described in Section 4.8 reduces the WCL of predictable cache coherence mechanisms by 94% for a 8-core system while trading off between 1%-6% average performance compared to existing coherence mechanisms.

4.10 Conclusion

In this work, we present a systematic approach to bridge the WCL gap between predictable cache coherence mechanisms and alternative data communication mechanisms. Our approach

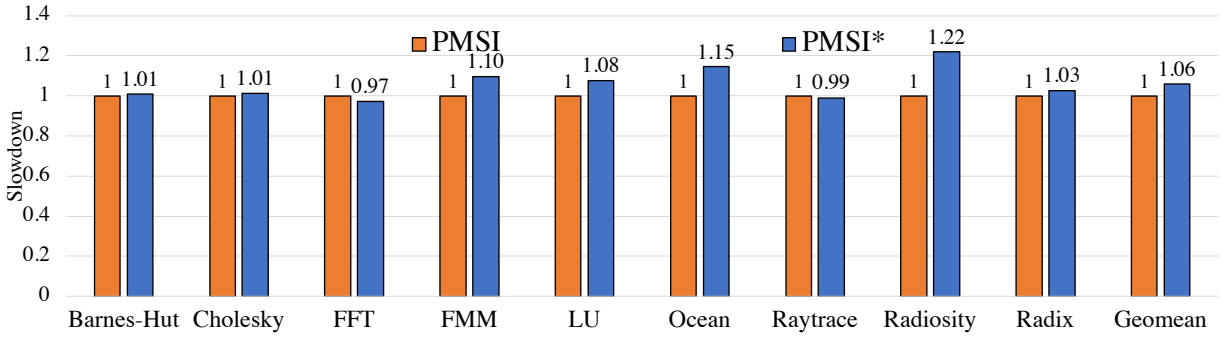


Figure 4.7: Slowdown of PMSI* on SPLASH-2.

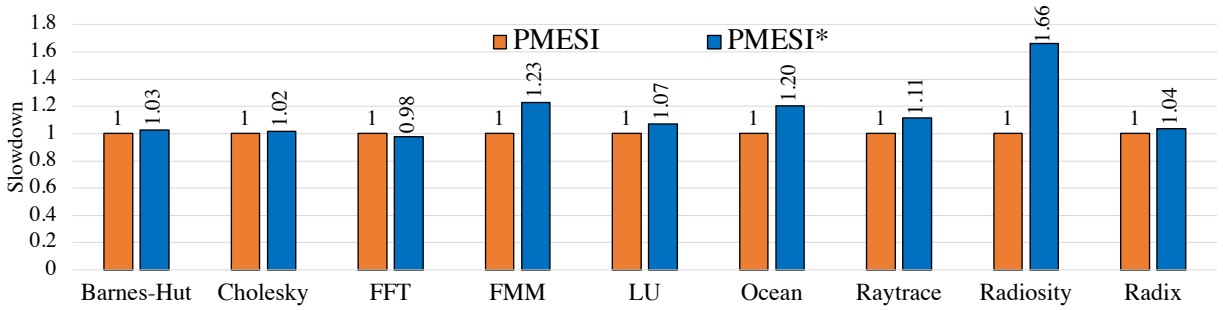


Figure 4.8: Slowdown of PMESI* on SPLASH-2.

consists of a formal framework that identifies the key reasons behind the high WCL in existing predictable cache coherence mechanisms. We describe one technique that tightens the WCL of predictable cache coherence mechanisms while retaining their performance advantage over alternative data communication mechanisms such as cache bypassing. Our proposed technique uses a combination of protocol changes and direct data communication through point-to-point data interconnects. We design two new predictable cache coherence protocols, PMSI* and PMESI*, using our proposed technique. Our evaluation shows that the WCL of a memory request under the new protocols is the same as that under cache bypassing, and hence, tighter than the existing PMSI and PMESI protocols. For a 8-core system, our proposed technique tightens the WCL of a memory request under the new protocols by 94% and trades away 1%-6% performance average performance compared to that under PMSI and PMESI protocols. Furthermore, the new protocols maintain their performance advantage over the cache bypassing mechanism (65% average performance speedup).

Chapter 5

Automatic Construction of Predictable and High-Performance Cache Coherence Protocols

Chapters 3 and 4 provided the tools that *guide* the design of predictable and high-performance cache coherence mechanisms. Recall that a cache coherence protocol is a component of a cache coherence mechanism and is implemented as a state machine that enforces the rules of coherent data communication. In Chapter 3, we implemented some of the design invariants in the cache coherence protocol resulting in the PMSI and PMESI cache coherence mechanisms and in Chapter 4, we showed that the design of the cache coherence protocol can significantly influence the performance and predictability guarantees. This makes the underlying cache coherence protocol a vital component of a hardware cache coherence mechanism.

Designing predictable and high-performance cache coherence protocols is a complex design exercise as it requires accounting for several communication scenarios between cores on a shared data. This is clear from the cache coherence protocol descriptions in Chapters 3 and 4. These coherence protocols have several t-states and transitions that are instrumental in achieving predictability and high-performance. To manage this design complexity, the *third research contribution* presents a tool, SYNTHIA, that automates the construction of predictable and high-performance cache coherence protocols. SYNTHIA refines an input specification of a cache coherence protocol that is devoid of any predictability and performance guarantees and outputs a complete cache coherence protocol implementation that guarantees predictability and performance.

5.1 Introduction

A hardware cache coherence protocol has a set of rules that ensures memory operations from cores operate on up-to-date versions of the requested data. The coherence protocol is a state machine with *coherence states,* and *transitions* between coherence states. Designing cache coherence protocols that deliver high-performance and that are correct is known to be challenging [107]. This is because the design process requires manually analyzing all possible interleavings of memory operations from different cores to the same shared data, and then constructing protocols that allow for these interleavings with little to no stalling of the memory operations. Designing one that also guarantees worst-case latency bounds (often called predictability) further exacerbates the challenge. This is because ensuring predictability while considering the many scenarios of interleaving memory operations across different cores requires intricate analyses of the hardware architecture and the protocol [74]. Missing one scenario can compromise predictability or limit the achievable performance.

The increase in complexity in designing predictable and high-performance cache coherence protocols comes in the form of additional states and transitions to the protocol [74, 138]. For example, the Modified-Shared-Invalid (MSI) protocol with no additional support for predictability or high-performance has 3 states and 12 transitions. A predictable and high-performance variant of the same protocol, however, has 15 states and 58 transitions [74]; a 5× increase in protocol size (number of states and transitions). A protocol designer is more prone to miss some states and transitions due to this dramatic increase in protocol complexity to achieve predictability and high-performance, which in turn compromises on correctness.

To improve productivity and simplify the construction of *correct, predictable, and high-performance* cache coherence protocols, we propose SYNTHIA, a tool that automates the coherence protocol construction. SYNTHIA takes as input a simple specification of a protocol that is devoid of states and transitions to achieve predictability or high-performance. This allows a protocol designer to focus on how a memory operation proceeds correctly *without* worrying about interleaving memory operations on the same data and carrying out the memory operation in a predictable manner. SYNTHIA *refines* this simple input specification and produces a predictable protocol implementation that achieves predictability and high-performance.

Our previous work described high level details about SYNTHIA's mechanism and applied SYNTHIA to three popular coherence protocols (MSI, MESI, and MOESI coherence protocols). In this work, we elaborate SYNTHIA's mechanism that constructs the protocol implementations and describe the construction of a predictable variant of a currently implemented coherence protocol. We extend our previous work [75] in two ways. First, we expand on SYNTHIA's mechanism that constructs the predictable and high-performance protocol implementation (Section 5.4). Our previous work [75] described the conditions under which new states and transitions were constructed, and did not describe the type of states and transitions constructed. Second,

we use SYNTHIA to construct a predictable variant of the Modified-Exclusive-Shared-Invalid-Forward (MESIF) protocol (Section 5.5), and evaluate the predictability and performance of PMESIF protocol using the gem5 micro-architectural simulator [17]. Recent reverse-engineering efforts by Sensfelder et al. [135] found that the NXP QorIQ multi-core platforms deployed the MESIF cacheh coherence protocol. We describe in detail the construction of the PMESIF protocol by SYNTHIA and highlight key predictability and performance features of the PMESIF protocol.

5.2 Main contributions

Our main contributions in this work are as follows:

- We present an approach to automatically construct predictable and high-performance snooping bus-based cache coherence protocols.
- We implement our approach in a tool called SYNTHIA. The input to SYNTHIA is a simple protocol specified in a domain-specific language SYNTHIADSL only using stable states. SYNTHIA uses the input protocol specification and carefully analyzes scenarios that require access to the shared bus including those that allow simultaneous interleaving memory operations on the same data. This analysis results in the construction of new states and transitions that achieve predictability and high-performance. The key operation in SYNTHIA is careful analysis of scenarios that require access to the shared bus, and allowing simultaneous interleaving memory operations on the same data to proceed without stalling.
- We evaluate SYNTHIA by generating predictable and high-performance protocol implementations for several common protocols such as the MSI, MESI, MOESI, and MESIF coherence protocol [135, 138]. On average, the complexity of the generated protocols have an increase of 4.9× the number of states and transitions. We thoroughly validate their correctness and ensure they are efficient using the gem5 micro-architectural simulator. SYNTHIA is available at https://github.com/caesr-uwaterloo/Synthia.

5.3 Related works

5.3.1 Predictable hardware cache coherence

Predictable hardware cache coherence protocols ensure that there is a worst-case latency bound on memory accesses across all cores [65, 67, 74, 77, 139]. These protocols are deployed on a

multi-core model that uses a shared snooping bus to communicate coherence messages between cores and the shared memory, and a shared data bus between cores and the shared memory. The shared snooping bus is a non-atomic split transaction bus [138]. The shared snooping bus and data bus deploy a *predictable arbitration policy* to predictably manage simultaneous accesses from cores. Examples of predictable arbitration policies include time division multiplexing (TDM) and round-robin (RR). These predictable arbitration policies divide access time to the shared bus into fixed time slots, and allocates these time slots to cores. A core is granted *exclusive* access to the bus at the start of its allocated slot. A core can only access the bus in its allocated slot; a pending bus access from a core that arrives immediately after the start of its allocated slot must wait for the start of its next allocated slot [74]. The memory hierarchy of the multi-core consists of one level of split private data and instruction write-back caches, and a shared last level cache memory. The private caches store a subset of data present in the shared memory. A core can communicate data in its private cache with other cores through point-to-point interconnects.

Prior works on designing predictable cache coherence protocols [65, 74, 77] modified existing conventional cache coherence protocols to satisfy predictability. These works first exhaustively analyzed different scenarios that can result in unpredictable scenarios. New t-states and transitions were constructed to address these unpredictable scenarios while maintaining data correctness and most of the performance benefits in the conventional protocols. Depending on the conventional protocol complexity, the analysis and the number of t-states and transitions to be constructed for predictability can be high making it an error prone process. SYNTHIA relieves this complexity burden by *automating* the analysis, and the construction of correct, predictable, and high-performance coherence protocols. A protocol designer provides SYNTHIA a protocol specification using only s-states and SYNTHIA automatically constructs a correct, predictable, and high-performance coherence protocol with the appropriate t-states.

Recently, Hessien and Hassan [67] described a new predictable bus architecture for snooping bus-based cache coherence mechanisms. A key feature of their work is that conventional cache coherence protocols can be deployed on this bus architecture with no protocol modifications, and the bus architecture guarantees predictable shared data communication between cores through cache coherence. On the other hand, the cache coherence protocols constructed by SYNTHIA are predictable through protocol changes and hardware structures in the cache controllers that work in tandem with the protocol changes [74].

5.3.2 Cache coherence protocol synthesis

Oswald et al. [107] presented ProtoGen, an automated tool that constructed high-performance directory-based cache coherence protocols. The input to ProtoGen was an atomic specification of a directory cache coherence protocol specified using stable coherence states. ProtoGen refined the input atomic specification by adding new transient states and transitions using domain

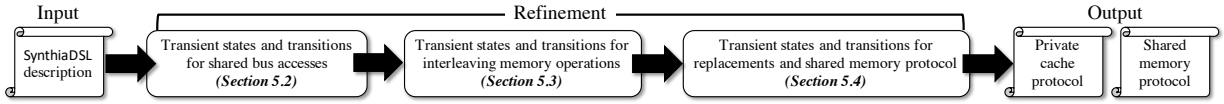


Figure 5.1: High level overview of SYNTHIA.

knowledge built into the tool. The output of ProtoGen was a non-stalling directory protocol implementation. While SYNTHIA takes inspiration from ProtoGen, it differs from it in two ways. First, SYNTHIA generates snooping bus-based coherence protocols, which have different designs and construction mechanisms compared to directory based protocols [138]. This is because of differences in coherence message communication (broadcast vs unicast) and ordering mechanisms (bus vs directory) [138]. This results in different protocol construction mechanisms. Second, SYNTHIA constructs *predictable high-performance* coherence protocols whereas ProtoGen constructed coherence protocols that only optimize performance. As a result, protocols generated with SYNTHIA can be used in real-time multi-cores. Furthermore, SYNTHIA is better positioned than ProtoGen for multi-core platforms with lower core counts (between 4-16 cores) where snooping-bus based protocols offer better average-case performance than directory-based protocols [138]. Recently, Oswald et al. extended their ProtoGen work with HieraGen [108] that constructed directory based protocols for multi-level cache hierarchies; ProtoGen constructed directory based protocols for a 2-level cache hierarchy. SYNTHIA constructs predictable cache coherence protocols for a 2-level cache hierarchy, and currently does not construct predictable cache coherence protocols for multi-level cache hierarchies.

Alternate protocol synthesis tools such as Transit [149] and VerC3 [35] relied on program synthesis that use a combination of designer provided guidance and model checking to complete partial descriptions of an input protocol specification. A key feature of these tools was frequent designer intervention to add information to the input specification for correct protocol construction [107]. We take an alternative approach by embedding domain knowledge about predictable high-performance snooping bus-based protocols into SYNTHIA to automate the protocol construction. On the other hand, Furthermore, SYNTHIA only requires a designer to provide a simple input specification, and generates the corresponding correct, predictable, and high-performance protocol implementation without further designer intervention.

5.4 SYNTHIA implementation

Figure 5.1 presents an overview of SYNTHIA. SYNTHIA takes as input a protocol specification written in SYNTHIADSL (Section 5.4.1). The specification consists of s-states and transitions between s-states at the private cache level. Note that SYNTHIA assumes the input specification is correct, and does not perform any verification for correctness on the input. The input

Table 5.1: Description of routines used for protocol construction.

Routine	Description
NEWTRANSIENTSTATE(x, y, z)	Construct new transient state of the form xy_z and sets state encoding of xy_z to be equal to x if pre-ordered t-state or y if post-ordered t-state.
NEWTRANSITION(x, y, e)	Construct new transition from state x to state y triggered on event e.
t.SOURCE()	Return the source state for transition t.
t.DESTINATION()	Return the destination state for transition t.
t.EVENT()	Return the triggering event for transition t.
t.SETDESTINATION$(()d)$	Sets the destination state of transition t to d.
ISOWN(ev)	Returns true if ev is an own event (OwnReadM, OwnRead, OwnWrite), false otherwise.
ISINVALID(s)	Returns true if access permissions of s is $invalid$, false otherwise.
ISDIRTY(s)	Returns true if data state of s is $dirty$, false otherwise.
ISACTIVE(s)	Returns true if data authority of s is $active$, false otherwise.
GETDST(s, ev)	Returns the destination s-state on applying event ev on source s-state s.
OWNTRANSITIONS(ev)	Returns the set of transitions triggered on other event ev. For example if ev is OtherWrite, then routine returns transitions triggered on OwnWrite.
ISCUMULATIVECHANGE(t_1, t_2)	Returns true if the cumulative data state or data authority across the source states in t_1 and t_2 are different from the cumulative data state or data authority across the destination states in t_1 and t_2, false otherwise.

is refined by creating new t-states and corresponding transitions, and results in a predictable and high-performance protocol implementation. This refinement identifies two main scenarios to construct t-states: (1) transitions that must wait for some communication on the shared bus such as broadcast coherence messages or data communication with shared memory (Section 5.4.2), and (2) transitions that change due to interleaving memory operations on the same cache line (Section 5.4.3). We explain the construction of transitions due to t-states using examples. After refinement, SYNTHIA outputs the cache coherence protocol state machines at the private cache level and shared memory. Table 5.1 describes the routines used in the protocol construction algorithms presented in the following subsections.

5.4.1 Protocol specification in SYNTHIADSL

The input information about s-states and transitions is defined in a domain specific language, SYNTHIADSL. There are two components in the input: (1) coherence state encoding of s-states and (2) transitions between s-states. Figure 5.2 shows the MSI protocol specification in SYNTHIADSL.

Coherence state encoding. Each s-state of a cache line specified in the input is a 3-tuple of the form (ap, ds, da) where $ap \in \{invalid, read, write, exread\}$ is the access permission, $ds \in \{clean, dirty\}$ is the data state of the cache line, and $da \in \{active, passive\}$ is the data authority of the cache line. The access permission conveys the type of memory operation (read/write)

```
1  M : (write, dirty, active)
2  S : (read, clean, passive)
3  I : (invalid, clean, passive)
4  (I, OwnReadM)     -> S
5  (I, OwnRead)      -> S
6  (I, OtherRead)    -> I
7  (I, OwnWrite)     -> M
8  (I, OtherWrite)   -> I
9  (S, Replacement) -> I
10 (S, OwnRead)      -> S
11 (S, OtherRead)    -> S
12 (S, OwnWrite)     -> M
13 (S, OtherWrite)   -> I
14 (M, OwnReadM)     -> M
15 (M, OtherRead)    -> S
16 (M, OtherWrite)   -> I
17 (M, Replacement) -> I
```

Figure 5.2: MSI protocol specification in SYNTHIADSL.

permitted on the cache line by a core. A core that does not have a cache line in its private cache has *invalid* access permissions. *read* denotes that a core can read the cache line data contents, and *write* denotes that a core can read and write the cache line data contents. Under the single-writer-multiple-reader (SWMR) invariant [138], at any instance of time only one core can have a cache line with write access permissions or multiple cores can have the cache line with read access permissions. *exread* denotes that a core can read the cache line data contents, and is the only core (exclusive) that has the cache line. The data state of a cache line conveys whether a core has modified the data contents of the cache line. A *dirty* data state means that a core may have modified the data contents, and *clean* data state means that the core has not modified the data contents. The data authority of a cache line conveys whether a core can communicate the cache line data contents to another core that requests for the same cache line via the point-to-point data interconnects. An *active* data authority means that a core can send the cache line data contents in its private cache to the requesting core and *passive* data authority means the core does not respond with data to another core's request. Lines 1-3 show the coherence state encoding for the **M**, **S**, and **I** states. For the MESI and MOESI protocols, the state encoding of **E** and **O** states are $(exread, dirty, active)$ and $(read, dirty, active)$ respectively. A key benefit of this state encoding is that protocols with states different from those found in the common MSI, MESI, and MOESI protocols can also be modeled in SYNTHIADSL. In Section 5.5, we show the construction of the predictable Modified-Exclusive-Shared-Invalid-Forward coherence protocol (PMESIF); MESIF protocol is deployed in NXP QorIQ multi-core processors [135].

Transitions. (src, ev)→dst is a transition where src is the source s-state, dst is the destination s-state, and ev ∈ {OwnReadM, OwnRead, OwnWrite, OtherRead, OtherWrite, Replacement} is the event that triggers the transition. OwnReadM event denotes a core's

own read request *issued* to a cache line, and the core receives the data response from the shared memory. OwnRead event denotes a core's own read request *issued* to a cache line, and the core receives the data response from its cache (cache hit) or from another core respectively. OwnWrite event denotes a core's own write request issued to a cache line, and the core receives the data response from the shared memory or its own cache (cache hit) or another core. OtherRead and OtherWrite events denote other cores' read and write requests to a cache line *ordered* on the bus. Replacement denotes a cache line replacement. Lines 4-17 in Figure 5.2 define the state transitions in the MSI protocol. For example, consider $(\texttt{I}, \text{OwnReadM}) \rightarrow \texttt{S}$. This means that a core performs a read operation on a cache line that it does not have in its private cache (I). On receiving the requested cache line data from the shared memory, the core transitions the cache line to S state. We use OwnWR (OtherWR) to denote a transition triggered on either OwnRead or OwnWrite (OtherRead or OtherWrite). Note that the MSI protocol has the same source and destination states for OwnReadM and OwnReadC events (I and S states). The MESIF protocol described in Section 5.5 has different destination states for OwnReadM and OwnReadC events.

Notice that the input SYNTHIADSL specification does not have (1) transitions triggered when a core observes its own memory operation, (2) transitions triggered when a core receives the requested data, and (3) actions that a core executes as a consequence of a transition such as sending data to another core (SD), and write-back data to shared memory (WD). These information are added by SYNTHIA during protocol construction. SYNTHIA automatically adds transitions triggered when a core's own memory operation is ordered on the snooping bus (Ordered) and on receiving the requested data (RD), and the appropriate actions based on the data state and data authority of the states involved in the transition.

The state encoding used in SYNTHIA is the same as that presented in Section 4.6 in Chapter 4. As a result, SYNTHIA also automates the analysis presented in Section 4.7 in Chapter 4, and computes the WCAL of the input cache coherence protocol specification. SYNTHIA identifies the offending transitions that results in the WCAL of a memory request under the input cache coherence protocol to be $\mathcal{O}(N^2)$ where N is the number of cores.

5.4.2 Constructing t-states and transitions due to shared bus communication

Key idea. Recall that the shared bus for predictable cache coherence protocols uses a predictable arbitration policy that allocates each core a fixed time slot to exclusively access the bus [74]. This means that a core must wait for its allocated time slot to communicate on the shared bus. These protocols use t-states to denote that a core has pending shared bus communication and is waiting for its allocated time slot [74]. Hence, SYNTHIA analyzes each transition in the input

Algorithm 1 t-states for shared bus communication

```
1:  procedure isTSNeededBusComm(t)
2:      src = t.source(), dst = t.destination(), ev = t.event()
3:      if isOwn(ev) then
4:          if src == dst then return false
5:          else if isInvalid(src) then return true
6:          else if isClean(src) ∧ ev == OwnWrite then return true
7:      else if isDirty(src) ∨ isActive(src) then
8:          tList = ownTransitions(ev)
9:          for all ot ∈ tList do
10:             if isCumulativeChange(ot, t) ≠ 0 then return true
        return false
```

specification, and identifies whether a transition must communicate coherence messages or data on the shared bus.

Mechanism. Algorithm 1 describes the conditions under which SYNTHIA constructs t-states and transitions for shared bus communication and Algorithm 2 describes the construction of t-states and transitions for shared bus communication. Algorithm 1 is used in Algorithm 3 in Section 5.4.3. The input to Algorithm 2 is a transition t, and it is applied to each transition in the input SYNTHIADSL specification. This algorithm exploits two key insights. First, for transitions triggered on own memory operations (line 3), t-states are required only when (a) `src` has *invalid* access permissions and `src` $\neq$ `dst` (lines 6-11) or (b) `src` has *clean* data state and the operation is OwnWrite (lines 12-15). Second, for transitions triggered on other memory operations, t-states are required depending on the overall state of the cache line before and after the memory operation *across all cores* (lines 16-28). New transitions are required for cases that introduce new t-states. Using Figure 5.3 as an illustrative example, we explain this implementation.

Consider insight (1). If `src` has *invalid* access permissions (ISINVALID returns true), then `src` does not have the cache line data contents to complete its own memory operation (lines 6-11). Hence, such transitions require t-states that wait for both the broadcast of coherence message regarding the memory operation to be ordered on the bus and the requested data contents. Lines 7 and 8 construct **_AD** and **_D** t-states, and lines 9-11 constructs the transitions due to these new t-states. In Figure 5.3, (**I**, OwnRead) $\rightarrow$ **S** has t-states **IS_AD** and **IS_D** where **IS_AD** waits for the coherence message broadcast to be ordered and **IS_D** waits for the requested data. The original transition (**I**, OwnRead) $\rightarrow$ **S** is changed to (**I**, OwnRead) $\rightarrow$ **IS_AD**, (**IS_AD**, Ordered) $\rightarrow$ **IS_D**, and (**IS_D**, RD) $\rightarrow$ **S**. For transitions (`src`, OwnWrite) $\rightarrow$ `dst` where `src` has *clean* data state, SYNTHIA also constructs **_AD** and **_D** states. Note that although `src` has the cache line data contents to complete its OwnWrite operation, receiving the cache line data contents before completing the OwnWrite operation in such a scenario simplifies the protocol design when taking into account interleaving memory operations from other cores to

Algorithm 2 Construction of t-states and transitions due to shared bus communication

```
1:  procedure CONSTRUCTTSANDTRANSITIONSFORBUSCOMM(t)
2:      src = t.SOURCE(), dst = t.DESTINATION(), ev = t.EVENT()
3:      if ISOWN(ev) then
4:          if ISINVALID(src) ∨ (ISCLEAN(src) ∧ ev == OwnWrite) then
5:              s1 = NEWTRANSIENTSTATE(src, dst, _AD)
6:              s2 = NEWTRANSIENTSTATE(src, dst, _D)
7:              t.SETDESTINATION(s1)
8:              t1 = NEWTRANSITION(s1, s2, Ordered)
9:              t2 = NEWTRANSITION(s2, dst, RD)
10:     else if ISDIRTY(src) ∨ ISACTIVE(src) then
11:         tList = OWNTRANSITIONS(ev)
12:         for all ot ∈ tList do
13:             if ISCUMULATIVECHANGE(ot, t) ≠ 0 then
14:                 s1 = NEWTRANSIENTSTATE(src, dst, _A)
15:                 t.SETDESTINATION(s1)
16:                 t1 = NEWTRANSITION(s1, dst, Ordered)
```

the same cache line [138].

Consider insight (2). Unlike the previous case, determining whether (`src`, Other) → `dst` requires t-states by solely looking at the properties of `src` and `dst` can introduce *unnecessary* t-states. Unnecessary t-states introduces unnecessary bus communication, which in turn causes unnecessary delays to the memory operation. As an example, consider the transitions (**M**, OtherRead) → **S** and (**M**, OtherWrite) → **I**. Although both **I** and **S** have same data authority and data state, (**M**, OtherWrite) → **I** does not require t-states whereas (**M**, OtherRead) → **S** requires at least one t-state. This is because (**M**, OtherRead) → **S** performs a write-back of the updated data contents, which must wait for the allocated time slot to communicate data to the shared bus. Hence, at least one t-state is required to indicate the pending write-back operation. On the other hand, (**M**, OtherWrite) → **I** does not require a write-back to shared memory, and the core that has the cache line in **M** can send the data to the requesting core.

We find that taking into account the *cumulative* coherence states of a cache line across *all* cores can identify whether (`src`, Other) → `dst` must access the shared bus, and hence, requires t-states. For example, consider a two-core system c_0 and c_1 where c_0 has cache line X in **M** state and c_1 does not have X (**I** state). Consider that c_1 issues an OwnWrite. c_0 moves to **I** and c_1 moves to **M** after c_1's OwnWrite based on the transitions described in Figure 5.2. Notice that only one core has X in **M** state before and after c_1's memory operation. Hence, the *cumulative* data state and data authority of X across all cores remains the *same* before and after c_1's memory operation. As a result, there is no need for c_0 to communicate the updated data contents of X to the shared memory. If c_0 performs a write-back to shared memory, c_0's updates to X will be overwritten by c_1's write operation to X. Furthermore, c_0 need not inform the shared memory about the change

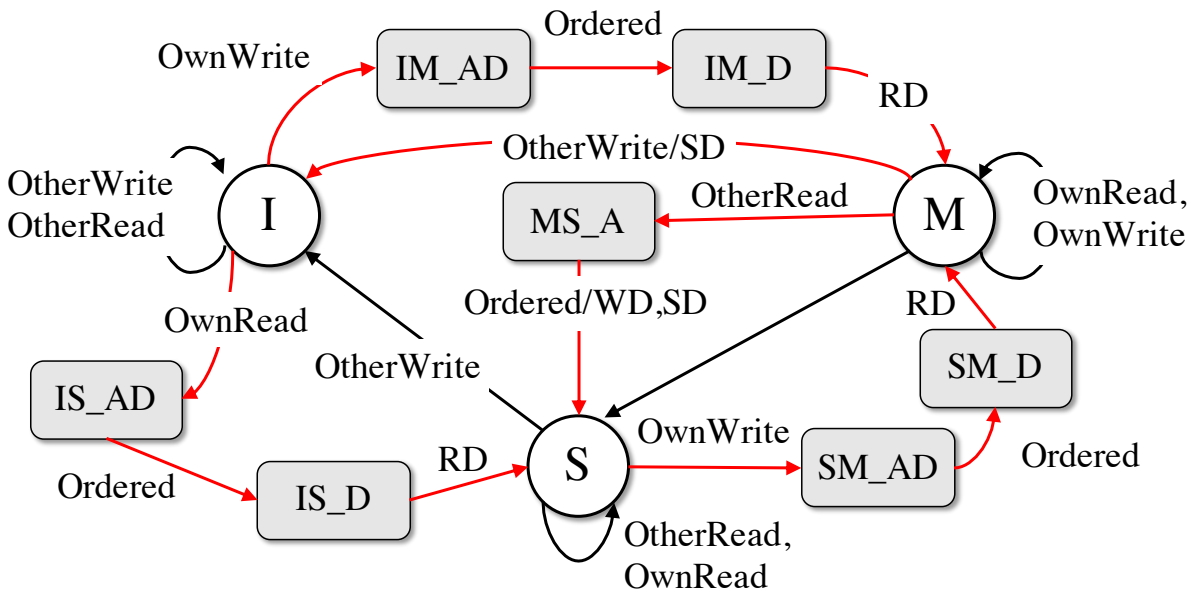


Figure 5.3: MSI protocol refinement for communication on the shared bus. Constructed t-states and transitions are highlighted.

in its data authority of X. This is because c_1 receives X with *active* data authority, and hence, c_1 responds to subsequent requests to X. Alternatively, consider that c_1 issues an OwnRead. c_0 and c_1 transition to **S** after c_1's OwnRead. In this scenario, the cumulative data state and data authority of X across all cores *changes* after c_1's OwnRead. Before the memory operation, c_0 has X with *dirty* data state and *active* data authority, and after the memory operation, c_0 and c_1 have X with *clean* data state and *passive* data authority. In this case, c_0 must communicate the change in data authority (from *active* to *passive*) and data state (*dirty* to *clean*) in X to maintain data correctness. In the MSI protocol, the communication of both data state and data authority changes are realized by c_0 doing a write-back to shared memory; the **M** state has *dirty* data state and *active* data authority. As a result, this scenario requires t-states. Note that conventional cache coherence protocols do not need t-states in this scenario. This is because conventional protocols are deployed on multi-core models where the shared bus does not allocate exclusive time slots to cores. Hence, cores can respond immediately on observing other memory operations on the bus, which removes the need for t-states [138].

In Algorithm 2, SYNTHIA only considers transitions triggered on other memory operations where `src` has either *dirty* data state or *active* data authority Transitions triggered on other memory operations cannot upgrade their data state from *clean* to *dirty* and data authority from *passive* to *active*. OWNTRANSITIONS(ev) returns a list of transitions triggered on own memory operation based on ev. For example, if ev is OtherWrite, then OWNTRANSITIONS(ev)

returns valid transitions triggered on OwnWrite. For each returned transition from line 11, SYNTHIA computes the change in cumulative data state and cumulative data authority between destination and source states in t and ot. ISCUMULATIVECHANGE first computes a value based on the data state and data authority of the destination states in ot and t and a value based on the source states in ot and t, and then returns the difference between the computed values. A non-zero difference means that a core must respond with some operation that requires shared bus access, and hence, requires at least one t-state; otherwise no t-states are required. In Figure 5.3, consider (**M**, OtherRead) $\rightarrow$ **S**. Line 11 returns ot =(**I**, OwnRead) $\rightarrow$ **S** and (**S**, OwnRead) $\rightarrow$ **S**. ot =(**S**, OwnRead) $\rightarrow$ **S** violates the SWMR invariant as one core has the cache line **M** and another core has the cache line in **S** simultaneously. Hence, the only valid ot is (**I**, OwnRead) $\rightarrow$ **S**. The source states in ot and t are **M** and **I** and the destination states in ot and t are both **S**. Line 19 returns `true` as the cumulative changes in data authority and data state are not zero, which results in constructing **MS_A**. The original transition (**M**, OtherRead) $\rightarrow$ **S** is replaced with (**M**, OtherRead) $\rightarrow$ **MS_A** and (**MS_A**, Ordered) $\rightarrow$ **S**. Since **M** has $dirty$ data state and $active$ data authority, the transition (**MS_A**, Ordered) $\rightarrow$ **S** is causes the core with the cache line in **MS_A** to write-back the modified data contents to shared memory (WD) and send the data to the requesting core. On the other hand, consider (**M**, OtherWrite) $\rightarrow$ **I**. The valid ot returned in line 11 is (**I**, OwnWrite) $\rightarrow$ **M**. In this case, the source states in ot and t are **M** and **I** and the destination states in ot and t are **I** and **M** respectively. Line 19 returns `false` as there are no cumulative changes in data authority and data state. Hence, this transition does not have t-states.

5.4.3 Constructing t-states and transitions due to interleaving memory operations

Key idea. In the protocol so far, there is no information regarding what a core must do when it has a pending operation on a cache line *and* observes interleaving memory operations from other cores on the same cache line. For example, consider a core that has a cache line in **IM_D** that is waiting to receive the requested data to complete its pending OwnWrite. Notice that there are no transitions in Figure 5.3 that determine what this core should do on observing OtherWrite or OtherRead on the same cache line. This scenario can occur as it may take several cycles for the core to receive the requested data during which multiple cores can perform operations on the same cache line while it is in **IM_D**. One solution is to *stall* any state changes to a cache line in a t-state until it transitions to its destination s-state. This solution trades simple protocol design for reduced performance as it introduces stalls. On the other hand, minimizing stalling while still maintaining predictability requires careful analysis of state changes due to interleaving other memory operations on a cache line in a t-state. In this step, we perform such analysis to con-

Algorithm 3 t-states for interleaving memory operations

```
1: procedure isTSNeededInterleavingMemOps(t, ts)
2:     src = t.source(), dst = t.destination(), ev = t.event()
3:     for all oev ∈ {OtherRead, OtherWrite} do
4:         if isPreOrdered(ts) then
5:             newDst = getDst(src, oev)
6:             if newDst ≠ dst then
7:                 if isOwn(ev) then
8:                     if isInvalid(newDst) then
9:                         s1 = newTransientState(newDst, dst, _AD)
10:                        newTransition(ts, s1, oev)
11:                    else
12:                        s1 = newTransientState(newDst, dst, _A)
13:                        newTransition(ts, s1, oev)
14:                else
15:                    s1 = newTransientState(newDst, newDst, _A)
16:                    newTransition(ts, s1, oev)
17:                    newTransition(s1, newDst, Ordered)
18:            else
19:                newTransition(ts, ts, oev)
20:        if isPostOrdered(ts) then
21:            newDst = getNewDst(dst, oev)
22:            if newDst ≠ dst then
23:                s1 = newTransientState(ts, newDst)
24:                newTransition(ts, s1, oev)
25:                nt =(dst, oev) → newDst
26:                if isTSNeededBusComm(nt) then
27:                    s2 = newTransientState(dst, newDst, _A)
28:                    newTransition(s1, s2, RD)
29:                else
30:                    newTransition(s1, newDst, RD)
31:            else
32:                newTransition(ts, ts, oev)
```

struct t-states and transitions that capture the correct order of state changes due to interleaving memory operations. SYNTHIA relies on Algorithm 2 to achieve this minimal stalling while still maintaining predictability.

Mechanism. For this analysis, we first classify t-states into two categories based on the *relative ordering* of other memory operations observed by a cache line on the shared bus: *pre-ordered* and *post-ordered* t-states. Algorithm 3 constructs t-states based on this classification. The algorithm takes as input a t-state (ts) and the transition on which this t-state lies on (t). For both categories, t-states are not required if there is no stable state change due to interleaving

other memory operations. For example, (I, OtherRead) → I and (S, OtherRead) → S do not need t-states as the stable states do not change due to interleaving memory operations.

Pre-ordered t-states. A cache line is in a *pre-ordered* t-state if the core's pending memory operation is *not* yet ordered on the snooping bus. _**AD** and _**A** t-states are pre-ordered t-states as they wait for the core's memory operation to be ordered on the snooping bus. For example, IM_AD and MS_A are examples of pre-ordered t-states. IM_AD waits for the coherence message for its write request to be ordered on the bus and the requested data, and MS_A waits for the coherence message for its data write-back to shared memory to be ordered on the bus. A cache line in a pre-ordered t-state observes interleaving memory operations from other cores on the bus (if any) *before* it sees its own memory operation ordered on the bus. As a result, a cache line in a pre-ordered t-state reacts to other memory operations (if any) *in the same way as if the cache line is in the source state*. For example, IM_AD, which lies on (I, OwnWrite) → M, reacts to other memory operations in the same way as I, and MS_A, which lies on (M, OtherRead) → S, reacts to other memory operations in the same way as M.

Lines 4-19 in Algorithm 3 describe the constructing of new t-states and transitions for a pre-ordered t-state. In line 5, SYNTHIA applies the other memory operation oev on the source s-state of the transition, and extracts the new destination s-state ($newDst$). A new t-state *may* be required to capture any state change ($newDst \neq dst$) depending on the transition type of t. If t is triggered on an own memory operation (lines 7-13), then t-states are required in order to capture the state change, and appropriate transitions to ensure the own memory operation ultimately completes. For example, consider the pre-ordered t-state SM_A, which reacts to other memory operations in the same way as S. On an OtherWrite, a cache line in S invalidates its data contents and moves to I state. Hence, a cache line in SM_A must transition to a t-state that conveys that the cache line data contents are invalid and an OwnWrite operation is pending. Remaining in SM_A state on an OtherWrite operation violates the SWMR invariant. On lines 8-10, _**AD** t-state is required as the interleaving memory operation invalidates the cache line data contents ($newDst$ has $invalid$ access permissions). Lines 15-17 handle interleaving memory operations that change the destination state of transitions triggered on other memory operations. For such cases, SYNTHIA creates a new _**A** t-state with $newDst$, and the corresponding transitions. For example, consider t-states MS_A, which lies on (M, OtherRead) → S. MS_A is waiting for the coherence message to complete its data write-back to shared memory. An OtherWrite on M transitions to $nextDst$ =I, which is different than the current dst (S). Hence, SYNTHIA constructs t-state II_A, and the transitions (MS_A, OtherWrite) → II_A and (II_A, Ordered) → I. SYNTHIA adds the send data action to the requesting cores (SD) to the transition (MS_A, OtherWrite) → II_A (not shown in Algorithm 3). Note that II_A on observing the ordered coherence message for the data write-back, which was issued when the cache line was in MS_A, simply transitions to I. For transitions that do not change state due to interleaving memory operations (line 19), SYNTHIA adds a self transition on t-state ts. For

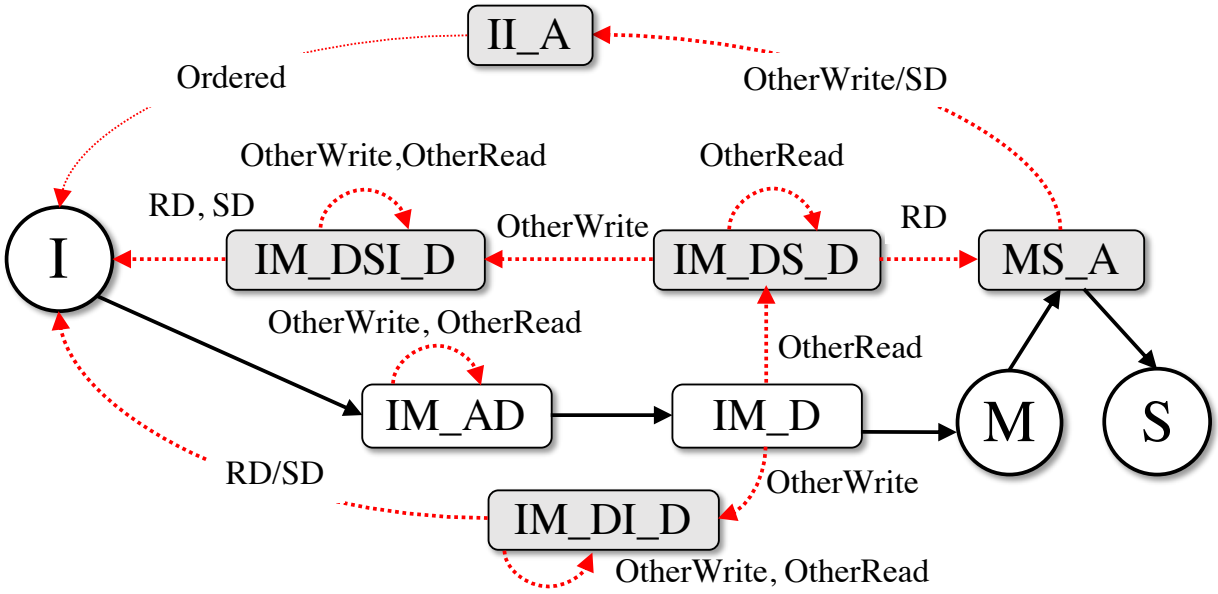


Figure 5.4: MSI protocol refinement for interleaving memory operations. Constructed t-states and transitions are highlighted.

example, interleaving memory operations on **IM_AD** behaves in the same way as **I**, which does not change state. Hence, SYNTHIA adds self transitions (**IM_AD**, OtherRead) → **IM_AD** and (**IM_AD**, OtherWrite) → **IM_AD**.

Post-ordered t-states. A cache line is in a *post-ordered* t-state *after* the core's pending memory operation is ordered on the snooping bus. Hence, other memory operations (if any) are ordered *after* the core's pending memory operation. A cache line in a post-ordered t-state reacts to other memory operations on the cache line *in the same way as if the cache line is in the destination state*. **_D** states are post-ordered t-states. For example, **IM_D** on (**I**, OwnWrite) → **M** and **IS_D** on (**I**, OwnRead) → **S** reacts to other memory operations in the same way as **M** and **S** respectively.

Lines 20-30 in Algorithm 3 construct t-states and transitions for post-ordered t-states. In contrast to pre-ordered t-states, SYNTHIA applies other memory operations on the destination s-state (line 21), and constructs t-states and transitions on a state change due to the other memory operations (lines 22-30). Consider **IM_D**. An OtherRead on **M** transitions to **S**. Hence, SYNTHIA constructs a new t-state **IM_DS_D** as shown in Figure 5.4, which captures the state change. This state conveys that a pending write operation on a cache line observed an OtherRead memory operation on the same cache line, and on receiving the requested cache line, the final s-state is **S**. A core that has a cache line in **IM_DS_D** completes the pending OwnWrite on receiving the requested data, and finally transitions to **S**. An OtherWrite on **M** transitions to **I**.

Similarly, SYNTHIA constructs a new t-state IM_DI_D on an OtherWriteas shown in Figure 5.4. On receiving data, IM_DI_D will transition to the stable state I. SYNTHIA applies Algorithm 3 on the new post-ordered t-states IM_DS_D and IM_DI_D. Since IM_DI_D lies between M and I, an other memory operation on IM_DI_D reacts in the same way as I. Hence, SYNTHIA creates a self transition for other memory operations on IM_DI_D as shown in Figure 5.4.

SYNTHIA uses Algorithm 2 to decide whether a post-ordered t-state transitions to the destination s-state directly or through other t-states. For example, consider a core that has a cache line in IM_DS_D. On receiving data, the core must complete the pending write operation, *write-back* the updated data contents, send the data to the requesting core, and transition to the final destination s-state S. Since, there is an operation that requires shared bus access (write-back), IM_DS_D cannot directly transition to S, and must first transition to a t-state (MS_A) to indicate pending write-back as shown in Figure 5.4. In this case, IM_DS_D lies on M and S, and ISTSNEEDEDBUSCOMM returns true for (M, OtherRead) → S (details in Section 5.4.2). Hence, SYNTHIA constructs t-states MS_A and the transition (IM_DS_D, RD) → MS_A to denote that on receiving the requested data, the cache line is marked for write-back. On the other hand, IM_DI_D can directly transition to I on receiving the data as ISTSNEEDEDBUSCOMM returns false for (M, OtherWrite) → I. This check for shared bus accesses in response to interleaving other memory operations, and the construction of t-states to indicate pending shared bus accesses is a key distinguishing feature between predictable coherence protocols and conventional cache coherence protocols.

5.4.4 Handling replacements, transition actions, and shared memory protocol construction

Replacements to cache lines with *dirty* data state or *active* data authority must write-back data to the shared memory or inform the shared memory regarding change in data authority respectively. For example, consider a replacement transition (s, Replacement) → i where s is a s-state with valid access permissions and i is a s-state with invalid access permission. If the data state of s is *dirty*, then SYNTHIA constructs a new t-state of the form si_A and transitions (s, Replacement) → si_A and (si_A, Ordered) → i. The transition (si_A, Ordered) → i is accompanied with write-back data to shared memory action. Similarly, SYNTHIA also constructs si_A and transitions (s, Replacement) → si_A and (si_A, Ordered) → i if the data authority of s is *active*. However, since the replacement changes the data authority of the cache line, the action accompanied with (si_A, Ordered) → i is a coherence message broadcast to inform the cores and shared memory the change in data authority of the cache line.

SYNTHIA also generates the protocol state machine at the shared memory. For any input protocol specification, the shared memory must maintain one state to denote if a cache line's data

contents are unmodified, and another state to denote if a core has a modified copy of the cache line in its private cache. Since a core may write-back updated data contents of a cache line to shared memory, the shared memory maintains a _**D** state that waits for the pending data communication from the core. Additional shared memory states are dependent on the s-states defined in the input specification.

Finally, for each transition in the core and shared memory protocols, SYNTHIA determines whether any action (SD, WD, BM) must be executed by a cache controller or shared memory on completing the transition. SD denotes that the cache controller or shared memory must send data to a requesting core, WD denotes that cache controller must send data to the shared memory, and BM denotes a coherence message broadcast. For a transition, the data state and data authority of the source state (s-state or t-state) denotes the type of action executed on exercising the transition. For example, a source state with $dirty$ data data state will perform a WD action on transitioning to a destination state with $clean$ data state.

5.4.5 Correctness of protocols constructed by SYNTHIA

In this section, we prove that the cache coherence protocols generated by SYNTHIA satisfy the SWMR invariant provided the input SYNTHIADSL specification satisfies the SWMR invariant. The SWMR invariant is a key correctness invariant for cache coherence protocols [138].

Theorem 5. *If the input* SYNTHIADSL *specification is correct, then the protocol implementation constructed by* SYNTHIA *is correct.*

Direct proof. A cache coherence protocol satisfies the SWMR invariant if and only if at any instance of time one or more cores have a cache line with read access permissions or only one core has the cache line with write access permissions. We make two observations from Algorithm 2 and 3 that show that SYNTHIA does not construct protocols that violate the SWMR invariant.

First, SYNTHIA does not remove s-states listed in the input SYNTHIADSL specification or add new s-states during the protocol construction; SYNTHIA only adds new t-states and transitions with these t-states. Hence, in the protocol constructed by SYNTHIA, multiple cores that have copies of a cache line in s-states satisfy the SWMR invariant. Second, the construction of t-states (lines 5, 6, 14 in Algorithm 2 and lines 9, 12, 15, 23, and 27 in Algorithm 3) are based on the transitions between s-states (src, dst, $newDst$) defined in the input correct SYNTHIADSL specification. SYNTHIA modifies the original transitions between s-states to include the new t-states but does *not* change the source and destination s-states of the transition. Since the state encoding of t-states is either the source s-state for pre-ordered t-states or destination s-states for post-ordered t-states, multiple cores that have the cache line in t-states also satisfy the SWMR invariant. Furthermore, putting these two observations together, multiple cores that have the cache line int-states and s-states also satisfy the SWMR invariant. □

5.4.6 Limitations of SYNTHIA

There are three main limitations of SYNTHIA. First, SYNTHIA assumes that the input protocol specification is correct, and does not verify or validate the correctness of the input protocol specification. Second, the final non-stalling protocol implementation is described in SYNTHIADSL. To empirically evaluate the constructed non-stalling protocol, a designer must convert the protocol implementation in SYNTHIADSL into an alternate implementation for micro-architectural simulation such as SLICC [17] or hardware logic implementation (Verilog or VHDL). Third, SYNTHIA generates snooping bus-based cache coherence protocols, and assumes all cores execute hard real-time tasks. As a result, SYNTHIA cannot generate predictable cache coherence protocols for mixed-critical systems deployed on multi-core platforms such as CARP [77] and Pendulum [139].

5.5 Case study: Predictable MESIF (PMESIF) cache coherence protocol

Recently, Sensfelder et al. [135] reverse engineered the hardware cache coherence protocol deployed in the NXP QorIQ multi-core processors. The NXP QorIQ multi-core processors are positioned for use in safety-critical systems such as avionics [106]. Their reverse engineering efforts revealed that the NXP QorIQ multi-core processor deployed a snooping bus-based MESIF cache coherence protocol. Their discovery is important as it shows that there exist multi-core processors used for safety-critical real-time systems that implement hardware snooping bus-based cache coherence mechanisms for shared data communication between cores. To satisfy the timing constraints of the deployed safety-critical tasks, the snooping bus-based cache coherence mechanism must be predictable [74]. For the NXP QorIQ multi-core processor, this means that the implemented MESIF cache coherence mechanism must be predictable. While we are not aware of all the implementation details of the MESIF cache coherence mechanism in the NXP QorIQ multi-core processors and hence, their predictability guarantees, we design one predictable variant of the MESIF cache coherence mechanism using SYNTHIA. The constructed predictable and high-performance variant of the MESIF cache coherence protocol, PMESIF, satisfies the design invariants listed in [74].

The MESIF protocol consists of 5 s-states. The **M**, **E**, **S**, and **I** states are the same as described in Section 5.4.1. The forwarding state **F** allows a core with a read only copy of the cache line to send the data to another core requesting for the same cache line. Hence, the state encoding of **F** is $(read, clean, active)$. The SYNTHIADSL specification of the MESIF protocol in stable states is shown in Figure 5.5.

We highlight one key feature about the MESIF protocol that makes it different from the MSI,

```
1  M : (write, dirty, active)
2  S : (read, clean, passive)
3  E : (exread, dirty, active)
4  F : (read, clean, active)
5  I : (invalid, clean, passive)
4  (I, OwnReadM)     -> E
5  (I, OwnRead)      -> F
6  (I, OwnWrite)     -> M
7  (I, OtherRead)    -> I
8  (I, OtherWrite)   -> I
9  (S, OwnRead)      -> S
10 (S, OwnWrite)     -> M
11 (S, OtherWrite)   -> I
12 (S, OtherRead)    -> S
13 (S, Replacement)  -> I
14 (M, OwnRead)      -> M
15 (M, OwnWrite)     -> M
16 (M, OtherRead)    -> S
17 (M, OtherWrite)   -> I
18 (M, Replacement)  -> I
19 (E, OwnRead)      -> E
20 (E, OwnWrite)     -> M
21 (E, OtherRead)    -> S
22 (E, OtherWrite)   -> I
23 (E, Replacement)  -> I
24 (F, OwnRead)      -> F
25 (F, OwnWrite)     -> M
26 (F, OtherRead)    -> S
27 (F, OtherWrite)   -> I
28 (F, Replacement)  -> I
```

Figure 5.5: MESIF protocol specification in SYNTHIADSL.

MESI, and MOESI cache coherence protocols. Consider the transitions $(\texttt{I}, \text{OwnRead}) \rightarrow \texttt{F}$ and $(\texttt{F}, \text{OtherRead}) \rightarrow \texttt{S}$. Notice that a core that has a cache line in **F** and observes an OtherRead not only sends the data to the requesting core but also changes the cache line coherence state to **S**; the core releases its *active* data authority on the cache line. As a result, the requesting core that receives data from another core (OwnRead), receives the cache line in **F** state. The benefit of transferring the **F** state of a cache line between requesting cores is that the likelihood of a cache line in **F** state to be a replacement candidate due to a cache capacity miss is lowered. A core that receives a cache line for its read request from another core will be the most recently used cache line in the cache set, and hence, will most likely not be a replacement candidate. Hence, the likelihood of cores receiving their requested cache line from other cores is high in MESIF protocol. On the other hand, the MSI, MESI, and MOESI cache coherence protocols do not have a forwarding state, and do not transfer data authority to cores performing read requests on cache lines.

Table 5.2 shows the private cache coherence states of PMESIF cache coherence protocol

Table 5.2: Private memory states for PMESIF cache coherence protocol generated by SYNTHIA. *issue msg/state* means the core issues the message *msg* and move to state *state*. Changes to conventional MESIF are in bold red.

	Core events			Bus events			
	OwnRead	**OwnWrite**	**Replacement**	**Receive data (RD)**	**Ordered**	**OtherRead**	**OtherWrite**
I	Issue OwnReadM/IS_AD	Issue OwnWrite/IM_AD				—	—
S	Hit, Complete read	Issue OwnWrite/SM_AD	-/I			—	-/I
M	Hit, Complete read	Hit, Complete write	Issue write-back/MI_A			Issue write-back/MS_A	Send data/I
E	Hit, Complete read	Hit, Complete write/M	Issue write-back/EI_A			Issue write-back/MS_A	Send data/I
F	Hit, Complete read	Issue write/FM_AD	Issue coherence message/FI_A			Send data/S	Send data/I
IS_AD					-/IS_D	—	—
IM_AD					-/IM_D	—	—
IS_D				If exclusive data from memory Complete read/E, else Complete read/F		—	-/IS_DI
IM_D				Complete write/M		-/IM_DS	-/IM_DI
IM_DS				Complete write, issue write-back/MS_A		—	-/IM_DSI
IM_DI				Complete write, send data/I		—	—
IM_DSI				Complete write, send data/I		—	—
IS_DI				Complete read/I		—	—
SM_AD			Stall		/SM_D	—	-/IM_AD
SM_D			Stall	Complete write/M		-/SM_DS	-/SM_DI
SM_DI			Stall	Complete write, send data/I		—	—
SM_DS			Stall	Complete write, issue write-back/MS_A		—	-/SM_DSI
SM_DSI			Stall	Complete write, send data/I		—	—
FM_AD			Stall		/FM_D	Send data/SM_AD	-/IM_AD
FM_D			Stall	Complete write/M		-/FM_DS	-/FM_DI
FM_DI			Stall	Complete write, send data/I		—	—
FM_DS			Stall	Complete write, issue write-back/MS_A		—	-/FM_DSI
FM_DSI			Stall	Complete write, send data/I		—	—
MI_A	Hit, Complete read	Hit, Complete write			Write-back to memory/I	—	Send data/II_A
MS_A	Hit, Complete read	Hit, Complete write	-/MI_A		Write-back to memory, send data/S	—	Send data/II_A
EI_A	Hit, Complete read	Hit, Complete write/MI_A			Write-back to memory/I	—	Send data/II_A
ES_A	Hit, Complete read	Hit, Complete write/MS_A	-/EI_A		Write-back to memory, send data/S	—	Send data/II_A
FI_A	Hit, Complete read	Stall			Write-back to memory/I	—	Send data/II_A
II_A	Stall	Stall			-/I	—	—

generated by SYNTHIA. The cells in red indicate t-states and transitions that differ from the conventional MESIF cache coherence protocol. Table 5.4 maps the PMESIF t-states and transitions constructed by SYNTHIA to Algorithms 2 and 3. As an example, consider the construction of (E, OtherRead) $\rightarrow$ ES_A. In the input MESIF protocol description in Figure 5.5, a core that has a cache line in E transitions to S on observing OtherRead. The state encoding of E and S are $(exread, dirty, active)$ and $(read, clean, passive)$ respectively. The requesting core on re-

Table 5.3: Shared memory states and transitions of PMESIF cache coherence protocol

State	Events from core			
	Read	Write	Replacement	Data from core
I	Send exclusive data to requesting core, update owner/M	Send data to requesting core, update owner/M		
M	Update owner/F_D	Update owner/M	-/I_D	
F_D	Update owner/F_D	Update owner/M	—	Write memory/F
I_D				Write memory/S
S	Send data to requesting core, update owner/F	Send data to requesting core, update owner/M		
F	Update owner/F	Send data to requesting core, update owner/M	-/I_D	

Table 5.4: PMESIF t-states and transitions constructed by SYNTHIA.

Algorithm lines	t-states and transitions constructed
Lines 4-9 in Algorithm 2	(I, OwnRead) → IS_AD, (I, OwnRead) → IM_AD, (IS_AD, Ordered) → IS_D, (IM_AD, Ordered) → IM_D, (IS_D, RD) → E/S, (IM_D, RD) → M, (S, OwnWrite) → SM_AD, (SM_AD, Ordered) → SM_D, (SM_AD, RD) → M, (F, OwnWrite) → FM_AD, (FM_AD, Ordered) → FM_D, (FM_D, RD) → M
Lines 10-16 in Algorithm 2	(M, OtherRead) → MS_A, (E, OtherRead) → ES_A
Lines 8-10 in Algorithm 3	(SM_AD, OtherWrite) → IM_AD, (FM_AD, OtherWrite) → IM_AD
Lines 11-13 in Algorithm 3	(FM_AD, OtherRead) → SM_AD
Lines 15-17 in Algorithm 3	(MS_A, OtherWrite) → II_A, (ES_A, OtherWR) → II_A, (MI_A, OtherWrite) → II_A, (EI_A, OtherWrite) → II_A
Line 19 in Algorithm 3	(SM_AD, OtherRead) → SM_AD, (IM_AD, OtherRead) → IM_AD, (IS_AD, OtherRead) → IS_AD, (IM_AD, OtherWrite) → IM_AD, (IS_AD, OtherWrite) → IS_AD, (ES_A, OtherRead) → ES_A, (MS_A, OtherRead) → MS_A
Lines 23-25 in Algorithm 3	(IM_D, OtherRead) → IM_DS, (IM_DS, OtherWrite) → IM_DSI, (IM_D, OtherWrite) → IM_DI, (IS_D, OtherWrite) → IS_DI, (SM_D, OtherRead) → SM_DS, (SM_D, OtherWrite) → SM_DI, (SM_DS, OtherWrite) → SM_DSI, (FM_D, OtherRead) → SM_D, (FM_D, OtherWrite) → FM_DI, (FM_D, OtherRead) → FM_DS, (FM_DS, OtherWrite) → FM_DSI
Lines 26-28 in Algorithm 3	(IM_DS, RD) → MS_A, (SM_DS, RD) → MS_A, (FM_DS, RD) → MS_A
Line 30 in Algorithm 3	(IS_DI, RD) → I, (IM_DI, RD) → I, (IM_DSI, RD) → I, (SM_DI, RD) → I, (SM_DSI, RD) → I, (FM_DI, RD) → I, (FM_DSI, RD) → I
Lines 31-32 in Algorithm 3	(IS_D, OtherRead) → IS_D, (IS_DI, OtherRead) → IS_DI, (IS_DI, OtherWrite) → IS_DI, (IM_DS, OtherRead) → IM_DS, (IM_DI, OtherRead) → IM_DI, (IM_DI, OtherWrite) → IM_DI, (IM_DSI, OtherRead) → IM_DSI, (IM_DSI, OtherWrite) → IM_DSI, (SM_DS, OtherRead) → SM_DS, (FM_DS, OtherRead) → FM_DS, (SM_DSI, OtherRead) → SM_DSI, (SM_DSI, OtherWrite) → SM_DSI, (SM_DI, OtherRead) → SM_DI, (SM_DI, OtherWrite) → SM_DI, (FM_DSI, OtherRead) → FM_DSI, (FM_DSI, OtherWrite) → FM_DSI, (FM_DI, OtherRead) → FM_DI, (FM_DI, OtherWrite) → FM_DI

ceiving the data transitions from I to F state. The input to Algorithm 2 is (E, OtherRead) → S, and this transition exercises lines 14-20 since ev =OtherRead. The condition on line 14 returns true as E has *dirty* data state and *active* data authority. OWNTRANSITIONS(OtherRead) returns (I, OwnRead) → F. The condition on line 17 returns true as there is a cumulative change in data state from *dirty* to *clean* across all cores when the cache line in one takes the transition (E, OtherRead) → S and the cache line in the requesting core takes the transition (I, OwnRead) → F. Hence, SYNTHIA constructs the new t-state ES_A, and transitions (E, OtherRead) → ES_A and (ES_A, Ordered) → S.

We highlight two key features of the PMESIF protocol.

Feature 1. Unlike the conventional MESIF coherence protocol where a core with a cache line in

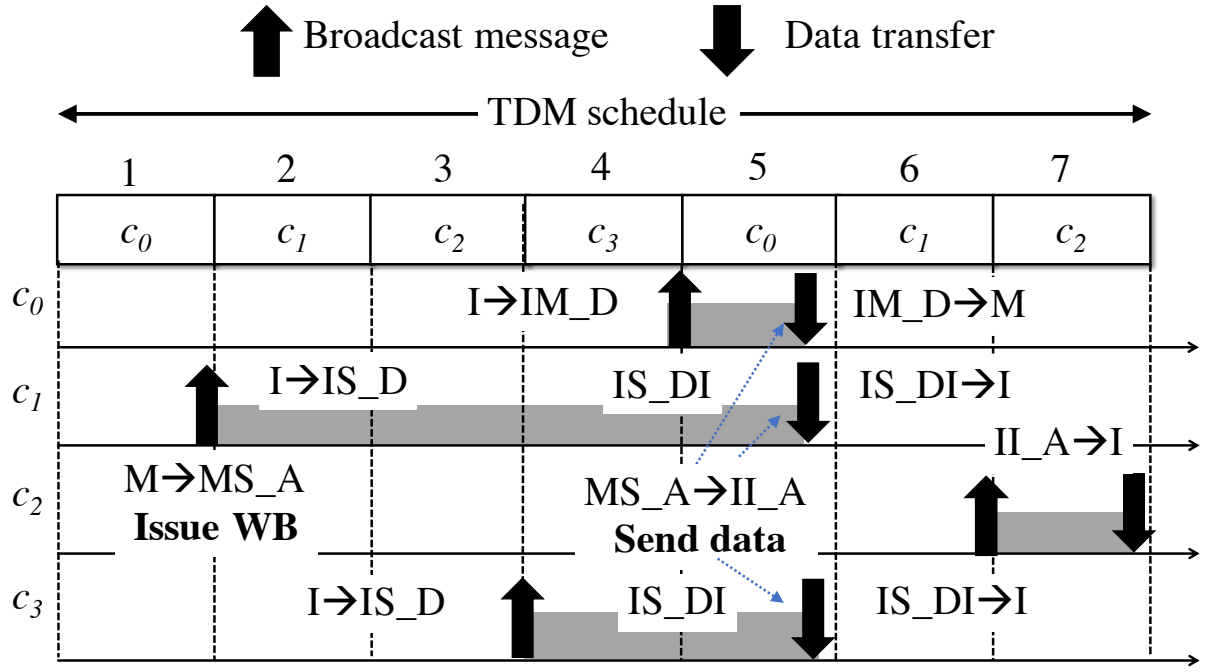


Figure 5.6: Execution example under PMESIF protocol.

M/**E** sends the requested data to *the first requesting core*, a core that has a cache line in **MS_A**/**ES_A** can send data to multiple requesting cores through the point-to-point data interconnects. This is because a core that has a cache line in **MS_A**/**ES_A** can observe multiple requests to the same cache line from different cores while waiting for its allocated slot to perform the write-back to shared memory. Figure 5.6 shows an execution example on cache line A accessed by 4 cores (c_0-c_3) where c_2 has A in **M** state. For this execution example, we assume the shared buses deploy a time division multiplexing (TDM) predictable bus arbitration, and each core is allocated one time slot. In slot 2, c_1 broadcasts a read request on A. This causes c_2 to issue a write-back coherence message and change the coherence state of its cache line copy of A to **MS_A**. For this example, assume that c_2 completes the data write-back in its allocated slot (slot 7). In slot 4, c_3 also broadcasts a read request on A. c_2 observes c_3's read request and does not change the coherence state of its A copy. c_0's write request to A causes A in c_2 to transition to **II_A**. c_2 sends the cache line data contents of A to the requesting cores c_0, c_1, and c_3. This is shown in Figure 5.6 where c_2 sends A to all the requesting cores in slot 5. Since each core is connected to every other core through point-to-point data interconnects, all the cores receive A in slot 5 and complete their pending memory requests on A.

Feature 2. In the conventional MESIF coherence protocol, a core that has a cache line in **IS_DI** completes the pending read request on receiving the data, and *sends* the data to another requesting core that has a pending write operation on the same cache line [135]. Recall that under the MESIF

coherence protocol, a core that has a pending read request to a cache line receives the cache line data contents with $active$ data authority (**F** state). For example, consider the execution example in Figure 5.6. Under the conventional MESIF coherence protocol, c_1 sends the data to c_0 on receiving A. On the other hand, under PMESIF coherence protocol, a core that has a cache line in **IS_DI** does not send the data to another core on receiving the requested data. This is because under PMESIF coherence protocol, a core that does not complete its read request to a cache line in its allocated slot means that there exists another core that has the same cache line in either **MS_A** or **ES_A** state and is waiting for its allocated slot to perform a write-back of the requested cache line to shared memory. In the execution example in Figure 5.6, cores c_1 and c_3 do not complete their read requests to A in their allocated time slots (slots 2 and 4 respectively) because c_2 has to complete the write-back of A to shared memory. **MS_A** and **ES_A** have the same state encoding as **M** and **E** states respectively, and hence, have $active$ data authority. On observing an OtherWrite, the core with cache line in **MS_A**/ **ES_A** will respond with data to the requesting cores. As a result, the core with cache line in **IS_DI** does not send data responses under PMESIF coherence protocol.

5.6 Results

Evaluation of SYNTHIA. SYNTHIA successfully constructs non-stalling and predictable coherence protocols from s-states specifications of MSI, MESI, MOESI, and MESIF protocols. The MESI and MOESI protocol specifications are derived from [138] and [1] respectively. Table 5.5 shows the number of states and transitions in the input and output. The MSI-P and MESI-P protocols differ from the MSI and MESI protocols in that *all* states have passive data authority. As a result, all data communication between cores in these protocols are through the shared memory. The predictable and high-performance protocol of MSI-P is the PMSI protocol described in [74]. A key takeaway is the significant increase in the number of states and transitions in order to achieve predictability and high-performance. For example, a predictable and high-performance MOESI implementation has more than 5$\times$ the number of states and transitions compared to the input specification. Hence, SYNTHIA relieves the design burden on a protocol designer by automating the analysis and protocol construction. We validated the protocols generated by SYNTHIA against manually implemented verified versions of the protocols. We found that the states and transitions in the protocols generated by SYNTHIA matched the manually implemented versions. We also checked their correctness, predictability, and performance through exhaustive testing using the gem5 simulator [17].

SYNTHIA is designed to improve the productivity of protocol designers by automating the exhaustive analysis required to construct predictable and high-performance protocols. To highlight this key utility of SYNTHIA, we perform the following experiment. Suppose a protocol

Table 5.5: Evaluation of SYNTHIA on different protocols. SYNTHIA took less than a few seconds to construct the protocols.

Protocol	Input		SYNTHIA output		Validation		Stalling transitions based on Algorithm 2		
	States	Transitions	States	Transitions	Correctness	Testing	Disabled	Only pre-ordered	Only post-ordered
MSI	3	14	17	66	✓	✓	12 of 36	4 of 39	8 of 48
MSI-P	3	14	17	64	✓	✓	14 of 39	4 of 39	10 of 51
MESI	4	20	26	96	✓	✓	18 of 51	6 of 57	12 of 72
MESI-P	4	20	26	95	✓	✓	22 of 57	6 of 57	16 of 78
MOESI	5	25	27	103	✓	✓	18 of 57	6 of 60	12 of 78
MESIF	5	25	35	118	✓	✓	22 of 63	6 of 66	16 of 84

Table 5.6: Predictability and performance evaluation.

Protocol	Predictability (WCL in cycles)		Performance
	Observed WCL	Analytical WCL	Speedup
MSI	3061	6450	3.45×
MESI	3214	6450	3.35×
MOESI	2019	6450	3.99×
MSI-P	6364	7250	1.72×
MESI-P	5964	7250	1.69×
MESIF	3741	7250	3.24×

designer manually designed the protocols listed in Table 5.5, and missed certain analyses that account for interleaving other memory operations to the same shared data (Algorithm 2). The missing analyses result in stalled transitions in the output protocol, which limit the performance of the cache coherence protocol. Consider a case where a protocol designer does not perform *any* of the analyses outlined in Algorithm 2. For the MSI protocol, we observed that 12 transitions out of total 36 transitions are stalling transitions, which constitutes more than 30% of the transitions (highlighted in Table 5.5). Across all protocols, we observed more than 30% of the transitions in the output protocols are stalling transitions. If a designer accounts for only one type of t-states (post-ordered or pre-ordered), then 9%-16% of the transitions in the constructed protocols are stalling transitions. On the other hand, protocols generated by SYNTHIA have *no* stalling transitions due to interleaving memory operations from other cores. In summary, a protocol designer can construct protocols where more than a third of all the transitions are stalling transitions by missing some scenarios. SYNTHIA addresses this by automating the analysis and construction as described in Section 5.4.

Predictability and performance evaluation. We manually converted the states, transitions, and actions in the generated protocols into the SLICC syntax, and evaluated them using the gem5 micro-architectural simulator [17]. The conversion was straightforward as it involved describing the states, transitions, and actions in the output protocol in SLICC syntax. We used the synthetic workloads from [74] and verified the data correctness of the protocols. We modeled a 8-core

multi-core platform where the shared bus deploys a TDM arbitration. Each core is allocated one TDM slot. Table 5.6 shows the maximum observed worst-case latency (WCL) experienced by a memory request under the predictable cache coherence protocols, and the average-performance speedup of the protocols compared to a cache bypassing technique. The cache bypassing technique achieves predictable data sharing between cores by disabling private caching of shared data. From Table 5.6, the observed WCL across all protocols are within their derived analytical WCL bound, and the generated cache coherence protocols outperform (as high as 3.94×) the cache bypassing technique while achieving predictability.

Among the predictable cache coherence protocols, the PMOESI cache coherence protocol offers the best performance speedup (3.99× average speedup over cache bypassing and 15% over PMSI, which is the next best cache coherence protocol). The key reason for this is that the PMOESI cache coherence protocol has the least number of transitions that require communication with the shared memory. In the PMOESI protocol, a core that has a cache line in **E** state must write-back the data contents to the shared memory on an OtherRead event. A core that has cache line in **M** and observes an OtherRead event does not trigger a write-back to shared memory. In this scenario, the core responds with the requested data and transitions the cache line from **M** $= (write, dirty, active)$ to **O** $= (read, dirty, active)$; ISCUMULATIVECHANGE() in Algorithm 2 returns false. On the other hand, in the PMESI and PMESIF protocols, a core that has a cache line in **E** or **M** states must write-back the data contents to the shared memory on an OtherRead event. As a result, the PMESI and PMESIF protocols have lower performance benefits compared to the PMOESI protocol, yet still maintain more than 3× average performance improvement over cache bypassing.

Verification of cache coherence protocols. We use the framework by Sensfelder et al. [133] to formally verify the correctness, liveness, and safety properties of the protocols generated by SYNTHIA. This framework uses the UPPAAL modeling framework. We changed the models in [133] to reflect our system model, and verified correctness (SWMR invariant, data value invariant), liveness (program termination), and safety (latency of each memory request is within the WCL bound) properties of the PMSI, PMSI-P, PMESI, PMESI-P, PMOESI and PMESIF protocols.

5.7 Conclusion

We present SYNTHIA, an automated tool for constructing correct, predictable and high-performance cache coherence protocols from simple protocol specifications. SYNTHIA automates the analyses that identifies scenarios involving interleaving memory operations from multiple cores to shared data and that require access to the shared bus. SYNTHIA refines the input protocol using the analysis by adding new states and transitions that achieve predictability and high-performance.

We apply SYNTHIA on multiple coherence protocol implementations found in existing multi-core platforms such as the MSI, MESI, MOESI, and MESIF cache coherence protocols. We validated the correctness, predictability, and performance of the protocols generated by SYNTHIA, and confirmed that the states and transitions in the generated protocols matched manually implemented versions.

Chapter 6

CARP: A Hardware Cache Coherence Mechanism for Multi-Core Mixed-Criticality Systems

The previous chapters (Chapters 3, 4, and 5) focused on designing predictable hardware cache coherence mechanisms for *hard real-time systems* where all tasks running on cores are safety-critical and have strict timing requirements. However, tasks deployed on multi-core real-time systems deployed in avionics and automotive domains have *varying* safety-critical levels that have different different timing requirements. For example, a multi-core platform deployed in a vehicle may execute quality-of-service (QoS) tasks that control the in-vehicle infotainment system (low safety-critical) while simultaneously executing tasks that perform object detection (high-critical). These real-time systems are referred to as *mixed-criticality systems* (MCS).

The *fourth research contribution* of this thesis presents CARP, a predictable hardware cache coherence mechanism for *mixed-criticality systems* (MCS). The key feature of the predictable hardware cache coherence mechanism described in this chapter is that it is *criticality-aware*. This means that the state transitions in the underlying coherence protocol are triggered based on the observed memory activity *and* the criticality levels of the tasks performing the memory activity. In this chapter, we make use of mixed-critical systems certification standards for automotive domains such as AUTOSAR and ISO-26262 to guide the design of CARP. We show that CARP prevents the data communication activity of low-critical tasks to affect the temporal behavior of high-critical tasks as mandated by the standards.

6.1 Introduction

Mixed-criticality systems (MCS) consist of tasks with varying safety requirements [14, 22, 72, 103]. These tasks are typically categorized into different criticality levels based on the severity of consequences of any deviation in their safety requirements [152]. The avionic and automotive domains have adopted several principles behind MCS as seen in standards such as DO-178C, ISO-26262, and AUTOSAR [152]. These domains continue to notice an increase in demands for complex and integrated functionalities whose implementations require interactions and communication between these complex functionalities [25, 60]. There is considerable interest in the community in using *multi-core platforms* to deploy such functionalities [20, 30, 105, 126, 128] for resource consolidation and cost reductions.

There are several prior research efforts in deploying MCS on multi-cores, but many of them assume that tasks do not communicate with each other [27, 47, 56, 63, 64, 78, 84, 166]. This assumption is *not representative* of practical systems. For example, Hamann et al. [60] described that communication is prevalent in automotive embedded applications deployed on multi-cores. Hence, it is not surprising that recent research efforts have attempted to support communication between tasks in MCS [25, 45]. We are encouraged by this trend to *explore predictable data communication mechanisms for MCS*.

We observe that prior efforts on designing data communication mechanisms for MCS have the following side-effects: (1) they *underutilize* the performance opportunities available on multi-cores, and (2) they *disallow* communication between critical and non-critical tasks [25]. For the first side-effect, consider Chisolm et al.'s [25] work, which allows communication between critical tasks, but these tasks must be executed on the *same* core of a multi-core platform. Using this approach, if all tasks were to communicate with a common task, then they would all have to be deployed on the same core. This would result in underutilization of the multi-core platform (effectively single-core utilization). We find that this side-effect is a result of *certain guidelines put forth by the standards or lack thereof*. Therefore, any data communication mechanism that complies with the standards may also suffer from the same side-effect. Consider the AUTOSAR standard that allows data communication between critical tasks as long as they reside in the same memory partition (§ 2.1.2.2 and § 2.1.2.4) [10]. However, AUTOSAR also mandates that tasks sharing a memory partition must be executed on the same core (§ 2.7) [9], which restricts the utilization of hardware parallelism offered by multi-cores.

For the second side-effect, to the best of our knowledge, there is limited guidance in the standards on the requirements of communication between critical and non-critical tasks. Consequently, the general approach taken has been to disallow communication between such tasks. However, we find that practical deployments can benefit from communication between non-critical and critical tasks. Examples include the use of non-critical tasks for run-time monitoring where tasks monitor the execution of critical tasks for data correctness [158, 170, 171], and qual-

ity management tasks [42] that improve the overall functioning and responsiveness of the MCS. Presently, one way to incorporate such non-critical tasks in MCS is to *elevate* them to be critical tasks, which would most likely impact the safety-critical requirements of critical tasks. Note that ISO-26262 allows non-critical tasks to co-exist with critical tasks in the same memory partition as long as the safety requirements of critical tasks are not violated (§ 2.7) [42]. To collect on the potential benefits of incorporating non-critical tasks, communication must be allowed with critical tasks, but it must be enabled carefully to ensure predictability, and without affecting the safety-critical requirements of critical tasks.

In this work, we develop a data communication mechanism for MCS that (1) leverages performance opportunities offered by multi-core platforms, and (2) allows communication between critical and non-critical tasks without violating the safety-critical requirements of critical tasks. In particular, we focus on the safety requirements that deal with the *temporal properties* (worst-case latency bounds) of critical tasks. To accomplish this, we cautiously and excitedly step beyond the constraints placed by the standards with the hope of fostering discussions on proposed extensions for future evolution of the standards. To motivate benefits of such proposed extensions, we design CARP, a data communication mechanism based on hardware cache coherence that uses these proposed extensions to enable communication between critical and non-critical tasks, and deploys them across multiple cores. A key novelty of CARP is that it *dynamically* handles the communication between tasks based on their criticality levels. As a result, CARP is a *criticality-aware* data communication mechanism. This criticality awareness property allows critical and non-critical tasks to communicate such that the temporal properties of critical tasks are not affected by non-critical tasks. Prior works that facilitate data communication between critical and non-critical tasks such as [139] do not dynamically adapt the communication, and hence, introduce some timing interference in the temporal bounds of critical tasks. From our evaluation, CARP improves average-case performance by 30% over prior state-of-the-art data communication techniques proposed for MCS and real-time systems.
Our **main contributions** in this work are as follows.

- We propose CARP, a criticality-aware cache coherence protocol that enables high performance data communication between critical and non-critical tasks while ensuring the non-critical tasks do not interfere with the worst-case latency bounds (WCL) of critical tasks.
- We present a latency analysis for CARP to derive the WCL bounds on data communication.
- We compare CARP against prior approaches for enabling data communication using synthetic and SPLASH-2 workloads [157]. We show that the observed data communication

latencies are within the WCL bounds, and CARP offers improved average-case performance over prior approaches for predictable data communication.

6.2 Motivation

We list two key guidelines defined in the AUTOSAR standard that govern the design of data communication mechanisms. The *first* guideline allows tasks of different criticality levels to reside in the same memory partition (§ 2.1.2.2 and § 2.1.2.4) [10]. A consequence of this guideline is that tasks can communicate through shared data [25, 60]. Therefore, tasks of any criticality level can communicate with each other through *shared data* resident on the same memory partition. The *second* guideline states that tasks sharing a memory partition must execute on the same core (§ 2.7) [9]. This guideline forces tasks communicating via shared data to reside on the same core. There are two key limitations that these guidelines impose for MCS deployments on modern and future multi-core platforms. (1) Limiting the number of cores that can be used based on data communication patterns of the application. As multi-cores continue to have large core counts, there would be considerable underutilization of hardware resources in deployments where tasks communicate. (2) Deploying tasks onto processing elements best suited for their execution is limited. Heterogeneous multi-core platforms with various accelerators [105] match the needs of modern applications, and has received recent attention for MCS [53, 62]. For example, certain machine-learning or vehicle tracking functionality may use a graphics-processing unit, and other computations may use real-time cores. However, if these functionalities communicate, then such heterogeneous platforms cannot be used since the tasks must reside on the same core. Note that AUTOSAR does provide explicit and implicit communication mechanisms. However, explicit communication typically results in lower performance due to lack of caching, and implicit communication sacrifices data consistency [60].

In an effort to address these limitations, we present two possible extensions for consideration. The first *extension* allows communicating tasks to be *deployed* across multiple cores provided the safety-critical requirements of critical tasks are not violated. Recently, Hassan et al. [74] proposed an approach to allow communicating tasks of the same criticality level to execute on different cores. They showed significant performance improvements over deploying communicating tasks onto the same core while preserving safety-critical requirements. Our work distinguishes itself from [74] in that we focus on data communication across tasks of different criticality levels deployed across multiple cores.

The second *extension* allows communication between critical and non-critical tasks such that non-critical tasks do not violate the safety-critical requirements of critical tasks. To the best of our efforts, we did not find any guidance in AUTOSAR for data communication between non-critical and critical tasks. Hence, the only way to allow such tasks to communicate is to

Table 6.1: AUTOSAR guidelines satisfied and extended by CARP.

CARP feature	Standard guideline	Relationship
Multiple tasks share a memory partition	§ 2.1.2 [10]	Satisfies
Tasks of different criticality levels share a memory partition	§ 2.1.2 [10]	Satisfies
Critical and non-critical tasks share a memory partition	§ 2.7 [42]	Satisfies
Data communication between non-critical and critical tasks	None	**Extends**
Tasks sharing a memory partition are deployed across cores	§ 2.7 [9]	**Extends**

elevate the criticality level of non-critical tasks. However, the introduction of newly elevated critical tasks will interfere with the temporal requirements of existing critical tasks. Hence, an alternative mechanism that allows for such communication without impacting the temporal requirements of critical tasks is desirable. Note that ISO-26262 does allow non-critical tasks to co-exist with critical tasks in the same memory partition as long as non-critical cores do not violate the safety requirements of critical tasks (§ 2.7) [42], but the general approach to such communication has been to disallow it. In this work, we *disallow* level E tasks to potentially corrupt memory contents by restricting them to only *read* communicated data.

Table 6.1 summarizes the relationship between CARP's features and the guidelines described in the AUTOSAR and ISO-26262 standards. This relationship falls into 2 categories: (1) *Satisfies*: CARP's features satisfy the guidelines in the standards and (2) *Extends*: CARP's features require extensions to the standards. We view CARP as a step towards identifying the benefits of extending the guidelines on data communication in order to develop high performance yet predictable data communication mechanisms for multi-core MCS.

6.3 System Model

We denote a task set with $\mathbb{T}$ tasks in the system as $\Gamma = \{\tau_i{}^l : l \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}, i \in [0, \mathbb{T}-1]\}$ where every task has a criticality level l [152]. Our MCS model follows standards in avionics and automotive domains that classify tasks into different criticality levels based on their safety requirements [10, 39, 42, 124]. A task may have one of the five criticality levels: level A tasks are the most critical whose failure may result in fatalities through level E that are not critical and experience system performance impacts on a failure. We collectively refer level A-D tasks as

critical tasks and level E tasks as *non-critical tasks*. Levels A and B tasks mandate tight WCL bounds with level A tasks having more stringent requirements than B. Tasks at levels C and D are soft real-time tasks that also need WCL bounds; however, these bounds are less stringent than levels A and B tasks. Level E tasks do not have any WCL bounds. Such a MCS model has been used in prior research [25, 78, 84, 99] [1].

Our real-time multi-core platform has N cores $C = \{c_0, c_1, ..., c_{N-1}\}$. A task mapped onto a core inherits the task's criticality level. For example, ${\tau_i}^l$ mapped onto core c_j results in c_j^l indicating that a task of l criticality level is executing on c_j. Note that we do *not* constrain a core to run a single task, and multiple tasks with different criticality levels can be executed on the same core. We require the core to identify the criticality level of the task currently executing on it. We describe one architectural extension to achieve this identification in Section 6.5. For the remainder of the text, we use the term core to refer to the task executing on the core. We denote $C^l = \{c_i^m : \forall c_i^m, m = l\}$ as the set of cores running tasks of criticality l. We assume that cores are in-order and allow for at most one outstanding memory request. Our evaluation empirically validates the analysis with this assumption. However, the proposed solution is independent of the core architecture, and works with out-of-order cores. The cores have a private memory hierarchy with caches and shared memory. The caches hold a subset of data stored in the shared memory, and the shared memory holds all the data needed by tasks running on the multi-core platform.

Cores communicate with the shared memory through a shared bus as the interconnect. A shared bus deploys an *arbitration policy* that manages the communication over the bus. The arbitration policies place constraints on when cores and memories are granted access to the shared bus for communication, and/or the amount of bus bandwidth made available for cores and memories. We deploy our proposed data communication mechanism on variants of time-division multiplexing (TDM) and round-robin (RR) arbitration policies. These arbitration policies have received considerable attention in the real-time community [27, 63, 66], and have been implemented in real-time platforms [118, 130].

6.4 High level overview of CARP

CARP delivers performance benefits by allowing (1) communicating tasks to be distributed across cores, and (2) communicated data to be cached in the private caches of cores. It follows the template of guidelines put forth by Hassan et al. [74], and includes new features specifically catered for multi-core MCS. There are two interference scenarios that arise due to communication to/from level E tasks on the WCL bounds of critical tasks: (1) interference scenario due to data responses from shared memory and (2) interference scenario due to write-backs.

[1] We are aware that ISO-26262 and AUTOSAR standards define level D as the highest criticality level and level A as the lowest criticality level.

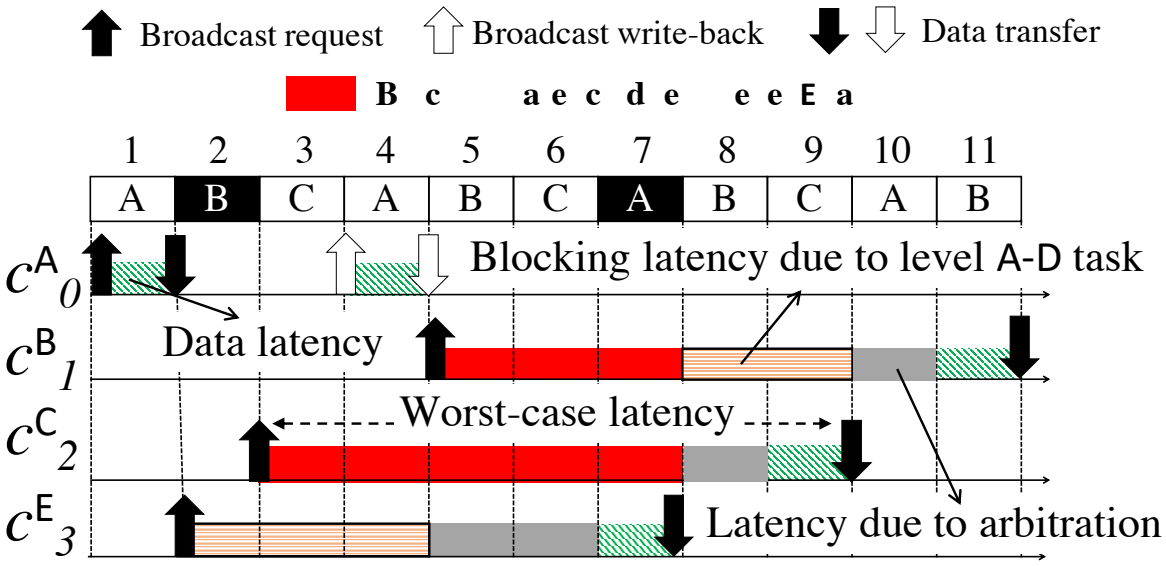


Figure 6.1: Blocking communication due to shared memory responses.

CARP disallows these interference scenarios through *two* techniques: (1) forces level E cores to *abort-and-retry* in the presence of simultaneous communication from critical cores on the same shared data and (2) *partitions* the PWB buffer to isolate write-back responses for critical cores from non-critical cores, and schedule write-back responses from non-critical cores in slack. In the following section, we illustrate these interference scenarios and the impact of the techniques using a 4-core system ($c_0 - c_3$) that executes a level A, B, C and E task respectively. For ease of explanation, we consider a schedule that uses TDM arbitration. This schedule allocates one slot for each levels A-C cores to manage concurrent accesses to the shared memory. Following the guidelines set by Mollison et al. [99], we use slack to schedule data communication to/from E tasks.

6.4.1 Interference due to data responses from shared memory

Observation. Figure 6.1 highlights the interference scenario due to data responses from shared memory. In the first slot, c_0^{A} broadcasts a write request to data X. The shared memory sends X in the same slot, and c_0^{A} completes its request. An updated copy of X resides in the private memory of c_0^{A}, and a stale copy of X resides in shared memory. The second slot, which is allocated to c_1^{B}, is unused by c_1^{B} making it a slack slot. Suppose c_3^{E} uses this slack slot and broadcasts a read request to X. Since, c_0^{A} has an updated version of X, it has to write back the update to the shared memory before the shared memory can send X to c_3^{E}. Hence, c_0^{A} must wait for its allocated slot

to update the shared memory (slot 4). In the third slot, c_2^{C} broadcasts a write request to X. In slot 5, c_1^{B} issues a write request to X, and will receive X from the shared memory after c_2^{C}. After c_0^{A} updates the shared memory with the updated X, the shared memory can send data to the cores waiting on X. To maintain *data correctness*, the *shared memory must send data in the order of requests* [74, 138]. For example, if the shared memory reorders data response to c_1^{B}'s request over that of c_2^{C}'s request then c_1^{B} will receive a value X updated only by c_0^{A} and not by c_0^{A} and c_2^{C}. As a result, c_1^{B} operates on incorrect X value, which compromises data correctness. Hence, the shared memory must first send data to c_3^{E} and then to c_2^{C} and c_1^{B}. This *blocking* of data communication from the shared memory to c_2^{C} and c_1^{B} due to c_3^{E} is highlighted in red in Figure 6.1.

Solution. One potential solution is to *prioritize* data responses from shared memory to critical tasks over data responses to level E tasks. However, we observe that prioritization can *indefinitely* defer the data responses to level E tasks, which limits level E tasks' effectiveness to MCS functioning. For the example in Figure 6.1, consider that the shared memory prioritizes responses to c_2^C and c_1^B over the response to c_3^E. Since c_1^B does a write operation, the memory and c_1^B move to the modified state (M) after sending and receiving X respectively. After c_1^B completes the write operation, two conflicting scenarios exist that prevent c_3^E to receive X: (1) the shared memory cannot send X to c_3^E as it must wait for c_1^B to write-back the updated X, and (2) c_1^B does not mark X for write-back as it does not observe the pending read from c_3^E, which was issued *earlier* than c_1^B's request. Hence, an alternative solution that does not indefinitely defer data responses to level E tasks is necessary.

Revisiting the above example, we observe that if c_3^E *aborts* its current request to X on observing remote requests to X from critical cores and *retries* its request to X after observing requests from c_2^C and c_1^B, then c_1^B observes the request from c_3^E and schedules the write-back response for X. The order of requests observed by the shared memory will be c_2^C, c_1^B, and c_3^E, and the shared memory can send the updated X to c_3^E after c_1^B completes its write-back. Hence, the *abort-and-retry* mechanism ensures that critical cores will *not* be blocked by data responses to level E cores and level E cores will receive their data responses. Note that this mechanism in CARP offers a *trade-off* between the value received by a level E core for a request to shared data and the freedom from blocking due to communication to/from level E cores on the WCL bounds of critical cores. In particular, a level E core may receive a more updated value of the requested data compared to the value of the data when the level E core broadcasted its first request to the requested data. We find this trade-off to be acceptable as ensuring no interference from non-critical cores to the temporal requirements of critical cores is a key safety requirements in MCS [42, 44].

6.4.2 Interference due to write-back responses

Observation. Hassan et al. [74] proposed the PWB data structure in the CC to *isolate* requests made by a core that miss in the private cache and write-back responses due to memory activity

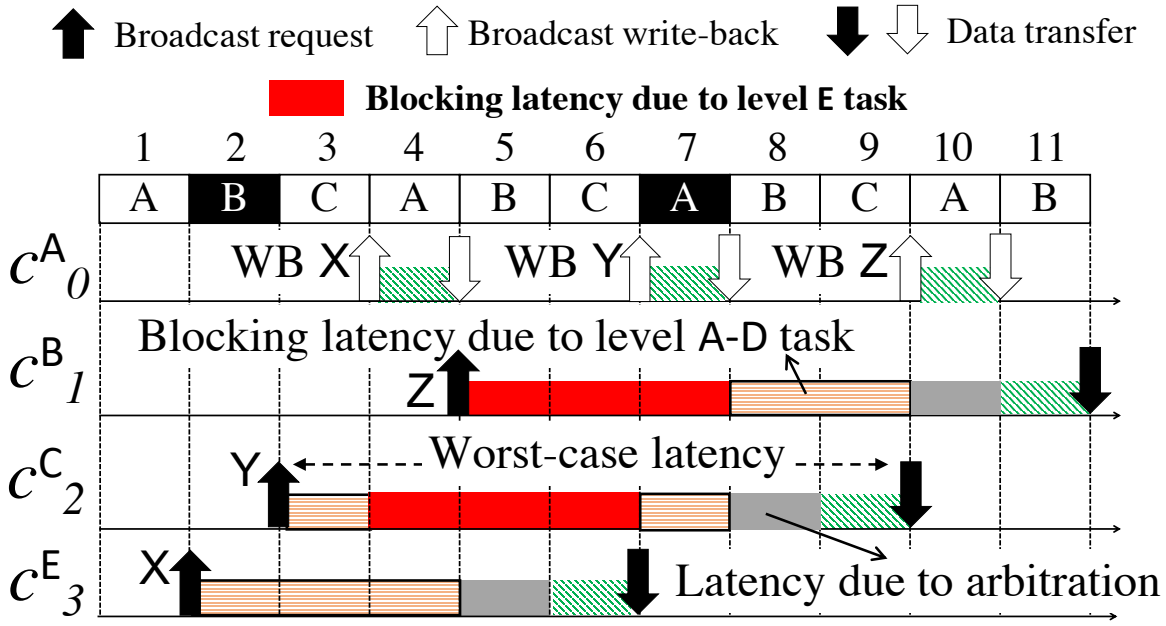


Figure 6.2: Blocking communication due to write-back responses.

from other cores [74]. They applied a predictable work conserving round-robin (RR) arbitration mechanism between the PR and PWB buffers as both cache miss requests and write-back responses require access to the shared bus [74]. However, their cache coherence protocol does not consider communication between mixed-critical and non-critical tasks [74]. Using Figure 6.2, we show that the current PWB design can cause blocking interference to critical cores in the presence of data communication between critical and non-critical cores.

Figure 6.2 modifies the example in Figure 6.1 such that c_3^E, c_2^C and c_1^B access different data blocks (X, Y, and Z respectively) that are modified by c_0^A and reside in c_0^A's private cache. Hence, c_3^E requests X in slot 2, c_2^C requests Y in slot 3, and c_1^B requests Z in slot 5. On observing these requests, c_0^A marks blocks X, Y, and Z for write-back by placing these blocks in its PWB buffer. Based on the guidelines listed in [74], write-back responses in a core's PWB are scheduled in a first-in-first-out (FIFO) order. Hence, in the first available slot of c_0^A that is marked for write-back, which is slot 4 in Figure 6.2, c_0^A will write-back X followed by write-backs to Y and Z in slots 7 and 10, respectively. Although the data requested by the critical cores are different from that requested by c_3^E, the FIFO order of draining the write-backs results in *blocking* communication to the critical cores as highlighted in Figure 6.2. In Figure 6.2, data response to c_2^C (Y) is blocked by the write-back response for c_3^E on X, which further blocks the data response to c_1^B (Z).

Solution. To eliminate this blocking interference, we *partition* the PWB of each core to isolate write-back responses for critical cores and non-critical cores, and schedule write-back responses

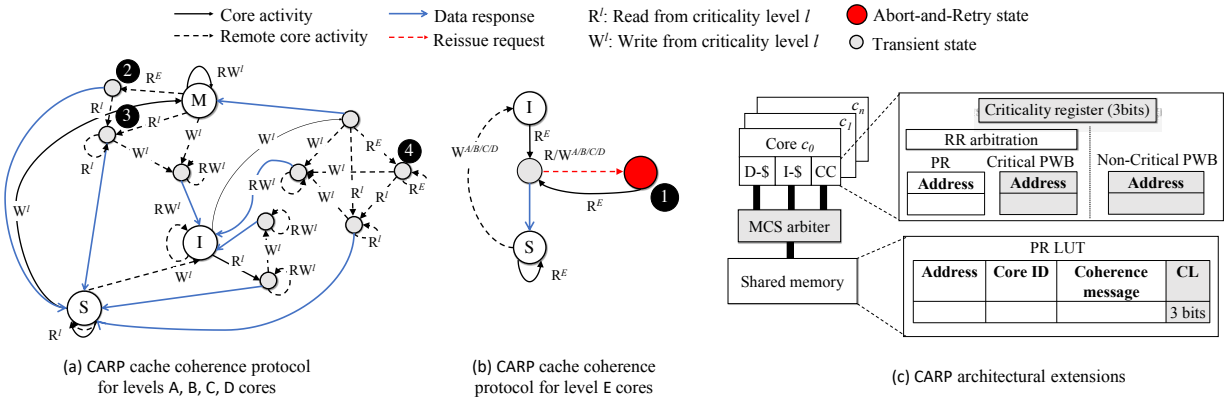
from the partition containing write-back responses to non-critical cores in slack. Applying this approach for the example in Figure 6.2, c_0^A will schedule the write-backs to Y and Z in slots 4 and 7 respectively, and does not incur any blocking interference from c_3^E's write-back response.

The rationale for using slack to schedule write-back responses for non-critical cores is based on the observation that implementing a cache coherence protocol for data communication *increases* the availability of slack in the system. This occurs because cores experience a larger number of hits in their private caches, which reduces the number of accesses to the shared memory. In turn, this reduces the utilization of the cores' allocated slots rendering them to be slack. Our evaluation shows that when using cache coherence, up to 40% and 76% of the allocated slots are unused for synthetic and real-world benchmarks rendering them as slack. Conventional slack allocation policies allocated slack to low criticality and non-critical cores that have pending requests [66,83,99]. In this work, we propose a different slack allocation policy that allocates slack for ready requests from low criticality and non-critical cores *and* pending write-back responses from non-critical PWBs across cores. Scheduling write-back responses due to non-critical cores in slack ensures no blocking interference due to write-backs on the WCL bounds of critical cores.

Note that CARP allows *bounded* timing interference on the timing guarantees of levels A-D cores due to other levels A-D cores. This bounded timing interference comes from the (1) predictable arbitration on the shared memory and (2) read-write memory activity of other levels A-D cores. Timing interference from (1) is a natural consequence when arbitrating accesses to a shared resource. The ISO-26262 standard suggests that tasks of different criticality levels can co-exist in the same memory partition as long as a lower critical task does not interfere with the timing requirements of a higher critical task (ISO-26262-9 §6.5) [42]. Given the examples of timing interference listed in ISO-26262 (Annex D in ISO-26262-6) [42], CARP does not allow for unbounded blocking of execution.

6.5 CARP implementation

In this section, we describe implementations of the techniques presented in Section 6.4. To implement abort-and-retry, level E cores must *differentiate* the criticality levels of requests broadcasted on the bus. Similarly, the PWB partitioning mechanism requires critical cores to *differentiate* between critical and non-critical requests to enqueue write-back responses in the appropriate PWB partitions. To this end, CARP's coherence protocol and architectural extensions enable CARP to be *criticality aware*. Figure 6.3 shows CARP's protocol state machine and the architectural extensions necessary to support CARP. CARP implements two coherence protocols: (1) for level A-D cores (Figure 6.3a) and (2) for level E cores (Figure 6.3b). Figure 6.3c shows the architectural extensions. Tables 6.2 tabulates the protocol state machine of CARP at the private cache level and shared memory level respectively. Table 6.4 describes the t-states introduced to sup-



(a) CARP cache coherence protocol for levels A, B, C, D cores

(b) CARP cache coherence protocol for level E cores

(c) CARP architectural extensions

Figure 6.3: CARP protocol specification.

port criticality awareness. In Table 6.4, we describe the events leading to the t-state (Cause field), and the operations executed by the CC in the t-state (Action field).

	Core events			Bus events						
	Read	Write	Replacement	OwnData	OwnUpg	OwnPutM	OtherGetS from A-D cores	OtherGetS from E cores	OtherGetM/OtherUpg	OtherPutM
I	Issue GetS/IS_D	Issue GetM/IM_D					—	—	—	—
S	Hit, Complete read	Issue Upg/SM_D	I/-				—	—	-/I	
M	Hit, Complete read	Hit, Complete write	Issue PutM/MI_A			Issue PutM/MS_A	PutM/MS_AE	Issue PutM/MI_A		
IS_D				Complete read/S			If E core, -/AR, else —	—	If E core, -/AR, else -/IS_DI	—
IM_D				Complete write/M			-/IM_DS	-/IM_DSE	-/IM_DI	—
SM_A			Stall		Complete write/M		—	—	Reissue write/I	
MI_A	Hit, Complete read	Hit, Complete write	—			Write-back to memory/I	—	—	—	
MS_A	Hit, Complete read	Hit, Complete write	-/MI_A			Write-back data to memory/S	—	—	-/MI_A	
MS_AE	Hit, Complete read	Hit, Complete write	-/MI_A			Write-back data to memory/S	-/MS_A	—	-/MI_A	
IM_DI				Complete write, issue PutM/MI_A			—	—	—	—
IS_DI				Complete read/I			—	—	—	—
IM_DS				Complete write, issue PutM/MS_A			—	—	-/IM_DI	—
IM_DSE				Complete write, issue PutM/MS_A			—	-/IM_DS	-/IM_DI	—

Table 6.2: Private memory states for CARP. *issue msg/state* means the core issues the message *msg* and move to state *state*. A core issues a *read/write* request. Once the cache line is available, the core *reads/writes* it. A core needs to issue a *replacement* to write back a dirty block before eviction. Changes to PMSI are highlighted.

State	Core events				
	GetS from A-D cores	GetS from E cores	GetM	PutM	Data from core
IoS	Send data to requesting core/IoS	Send data to requesting core/IoS	Send data to requesting core/M	—	
M	Clear owner/IoS_D	Clear owner/IoS_D	Update owner to requesting core/M	Clear owner/IoS_D	
IoS_D	Cancel pending level E core requests, Stall	Stall	Stall	Stall	Write memory/IoS

Table 6.3: Shared memory states for CARP protocol.

Table 6.4: t-states and transitions introduced in CARP.

t-state	Cause	Action
AR (❶)	A level E core waiting for data response observes a remote read/write request from a critical core.	Level E core aborts and retries its request.
MS_AE (❷)	A critical core that has modified data in private cache observes a remote read request from level E core.	The critical core enqueues write-back response in non-critical PWB.
MS_A (❸)	A critical core that has modified data in private cache observes a remote read request from another critical core.	The critical core enqueues write-back response in critical PWB and removes any matching entry in non-critical PWB.
IM_DSE (❹)	A critical core waiting for data response to complete write operation observes remote read request from level E core.	After completing write on data response, core enqueues write-back response in non-critical PWB.

6.5.1 Implementing abort-and-retry for level E cores

CARP introduces a *criticality register* in each core's cache controller that tracks the criticality level of the current task scheduled for execution by the real-time operating system (RTOS) as shown in Figure 6.3c. This allows the cache controller to broadcast the criticality level. Initializing the contents of the criticality register can be done either by the RTOS scheduler, or by the task through software extensions prior to a task's execution. The cache controller (CC) looks up the criticality level in this register when generating coherence messages.

Figures 6.3a and 6.3b show the protocol specifications to support *abort-and-retry* memory requests. For each request generated by a core, the core's CC looks up the contents of the criticality register and broadcasts the request along with the core's criticality information. For critical cores, the CCs broadcast read (R) and write requests (W) as R^l and W^l respectively where $l \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$ based on the criticality register contents. For non-critical cores, the CCs broadcast read requests as R^{E}.

Figure 6.3a shows CARP's protocol specifications for data communication between critical tasks in which all critical cores follow the *same* transitions and state changes for data communication between critical cores. On the other hand, Figure 6.3b shows CARP's protocol specifications for data communication between level E cores and critical cores in which different transitions are exercised based on the criticality levels of the remote requests. We revisit the example in Section

6.4.1 to highlight the state transitions for c_3^{E} in Figure 6.3b and for cores c_2^{C} and c_1^{B} in Figure 6.3a. c_3^{E} broadcasts its request to X, and transitions from I to a transient state that denotes c_3^{E} is waiting on data from the shared memory. The shared memory records c_3^{E}'s pending request in the PR LUT. c_0^{A} observes the level E remote read request on the shared bus, and updates its non-critical PWB with a pending write-back response to X. c_0^{A} then transitions from $M \rightarrow$ ❷. While c_3^{E} is waiting for the data response, c_2^{C} broadcasts a request to X. Due to this broadcast, two state transitions are exercised: (1) c_0^{A} observes a critical read request and transitions from ❷ $\rightarrow$ ❸ (Figure 6.3a) and updates its critical PWB with a pending write-back response to X and (2) c_3^{E} observes a critical read request and moves to the transient state denoted as ❶ in Figure 6.3b. At transient state ❶, c_3^{E} *aborts* its current request, and *retries* the read request to X by regenerating the read request.

Aborting a request requires discarding *all* information about the request from the system. In c_0^{A}, there are two pending write-back responses to X: (1) in the non-critical PWB due to c_3^{E}'s request and (2) in the critical PWB due to c_2^{C}'s request. At the PR LUT, there is an entry corresponding to requests from c_3^{E} and c_2^{C}. Hence, the aborting mechanism discards the write-back response in c_0^{A}'s non-critical PWB due to c_3^{E}'s request and the PR LUT entry corresponding to c_3^{E}'s request. Discarding entries in the non-critical PWB is done during the transition between transient states ❷ to ❸. Prior to updating the critical PWB, the CC scans the non-critical PWB for write-back responses to the same data, and discards them. CARP extends the PR LUT with criticality information and introduces logic in the shared memory controller that discards entries for level E cores in the PR LUT when the shared memory observes critical cores' data communication on the same data.

The above set of transitions for c_3^{E} repeat on observing c_1^B's remote request to X. Hence, the final order of requests observed by the cores and shared memory to X is: c_2^{C}, c_1^B and c_3^{E}. If there are no other requests to X from other critical cores between c_3^{E} request and its response from shared memory, X in c_3^{E} transitions to S on receiving X from the shared memory. Note that in the presence of critical requests to the same shared data, E-cores can continuously abort and retry their memory requests. We allow for this as temporal bounds are not required for level E cores.

6.5.2 Implementing PWB partitioning and slack scheduling for non-critical write-back responses

To address the blocking interference due to a single per-core PWB buffer (Section 6.4.2), CARP partitions the PWB into a *critical-PWB* and a *non-critical PWB* as shown in Figure 6.3c. Critical PWB enqueue write-back responses due to requests from critical cores, and non-critical PWB enqueue write-back responses due to requests from non-critical cores. For the example in Figure 6.2, the critical-PWB of c_0^A will have write-back responses for Y and Z, and the non-critical

PWB of c_0^A will have write-back response for X. To ensure that critical cores are not blocked by write-back responses due to level E cores, the arbitration policy between requests and write-back responses is only applied to the PR buffer and *critical-PWB* partition as shown in Figure 6.3c.

At the start of a slack slot, we prioritize ready requests from levels C-D cores over write-back responses and requests from level E cores as levels C-D cores execute higher criticality tasks. If there are no ready requests from levels C-D cores, the critical and non-critical PWBs of all cores are checked, and a write-back request is scheduled if found. If no ready requests from levels C-D cores and write-back responses in the cores are found, the slack slot is allocated to a level E core. We use RR to arbitrate across requests from multiple level E cores.

6.5.3 Hardware overhead

CARP protocol specifications and architectural extensions implement the three techniques described in Section 6.4. Compared to the prior predictable cache protocol proposed by Hassan et al. [74], CARP is criticality aware, and has additional transient states and transitions to eliminate the blocking latency of level E cores on the WCL bounds of critical cores. CARP introduces 3-bits of additional hardware storage per core to store the criticality information of the task executing on the core, and a 3-bit field in the PR LUT to support discarding of non-critical requests. Hence, for an 8-core system, the hardware overhead of CARP is 27-bits.

6.6 Latency analysis

We derive the per-request worst-case latency (WCL) bound a core experiences when it accesses data. The WCL bound of the requesting core has three latency components: (1) latency to broadcast data request on the network (request latency), (2) latency for a remote core with an updated copy of the requested data and/or the shared memory to place the data response on the network (communication latency), and (3) the latency of the data response to arrive at the requesting core (response latency). The arbitration scheme determines the request and response latencies. The communication latency, however, depends on the simultaneous communication between other cores in the system on the requested data. The WCL bound is the summation of these latency components.

6.6.1 Preliminaries

We envision CARP to be deployed on prior MCS-specific arbitration schemes such as [27, 63] that either combine different arbitration policies (TDM and RR) [27] or allocate different number of slots based on the core's criticality level (weighted TDM) [63]. The key features of these

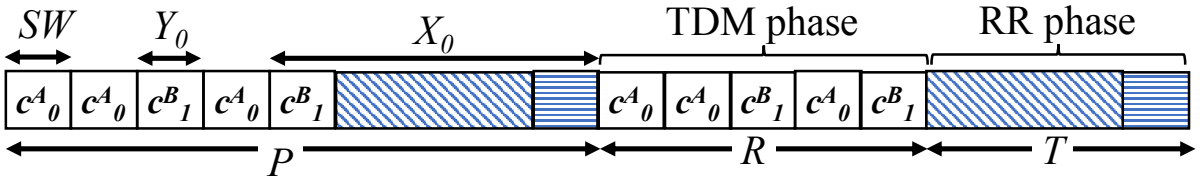


Figure 6.4: Generalized MCS arbitration scheme [27].

arbitration schemes are: (1) differential service guarantees to cores based on their criticality levels [27,63], and (2) slack allocation for lower critical tasks [25,66]. We capture these features in a representative arbitration scheme (shown in Figure 6.4), and use this scheme to derive the WCL bounds. Computing details such as the allocation of slots to cores is beyond the scope of this work; hence, these details are abstracted as arbitration parameters in the latency analysis.

The generic arbitration scheme consists of two arbitration phases: a weighted TDM arbitration phase (TDM-phase), and a RR arbitration phase (RR-phase). Levels A and B cores (C^{AB}) access the bus using TDM arbitration policy, and levels C and D cores (C^{CD}) access the bus using RR arbitration policy. We set the TDM slot width SW to be equal to completing one memory access. This slot width takes into account the latency to communicate coherence messages and data between cores and shared memory via the shared bus. We assume that the allocation of TDM slots to cores (arbitration schedule) is computed off-line. We denote s_i as the number of slots allocated to c_i^l in the weighted TDM schedule. For example, $s_0 = 3$ and $s_1 = 2$ in Figure 6.4. For a core c_i^l, we use X_i to mean the maximum number of slots between its next dedicated slot, and Y_i as the *next* maximum number of slots between its next dedicated slot. A RR-phase consists of a sequence of time slots that are distributed to cores in a work-conserving RR manner. Once a core is granted access to the bus in a RR-phase, it relinquishes access when the request is completed or a threshold amount of time (SW) elapses when the core is granted access to the bus. We denote R as the number of slots between two RR-phases as shown in Figure 6.4. A RR-phase is augmented with a *reserve time slot* to avoid bus interference between cores accessing the bus in RR-phase and TDM-phase phases [27]. In-flight requests that accessed the bus in a RR-phase are allowed to complete in the reserve slot. However, new requests from cores are not allowed to access the bus in the reserve slot. The length of a RR-phase, which includes the reserve slot, is denoted as T. The key difference between slots in the RR-phase and TDM-phase is that cores can access the bus at any time instance in a RR-phase, whereas cores access the bus at the start of a slot in a TDM-phase. The arbitration period is denoted as P. Level E cores are granted access to the bus in slack. We denote the latency to transfer data from the shared memory to the core as L^{acc}. Table 6.5 summarizes the symbols used in the latency analysis.

Note that CARP is not only designed for MCS-specific arbitration schemes, and can also be deployed on a simple TDM arbitration scheme that allocates equal number of slots to all cores in

Table 6.5: Symbols used in latency analysis.

Symbol	Description
SW	Slot width
c_i^l	Core i running level l task
s_i	Slots allocated to core c_i^l
X_i	Slots between two allocated slots of c_i^l
Y_i	Slots between two allocated slots of c_i^l
T	Length of RR-phase (includes reserve slot)
R	Slots between two RR-phases
P	Arbitration period

the system. However, the latency analysis for levels C-D cores, and their impact on the analysis of levels A-B cores will change as the analysis in the following section assumes RR allocation for levels C-D cores.

6.6.2 Analysis

First, we derive the worst-case request and data response latencies of a request issued by a core under analysis c_{ua} based on its criticality level (Theorems 6-7). Then, we present a critical instance that results in the worst-case communication latency incurred by c_{ua}'s request (Lemma 10). The critical instance identifies the memory access pattern of cores of different criticality levels that interfere with the data response of c_{ua}'s request to X. From this critical instance, Lemma 11 derives the worst-case number of cores and their criticality levels that interfere with c_{ua}'s request, and Theorems 8-9 derive the worst-case communication latencies of c_{ua} based on the criticality levels of the interfering cores. The WCL bounds are computed for criticality levels A-D. CARP does not provide any WCL bounds for level E cores; hence, we do not derive their WCL bounds.

Theorem 6. *The worst-case request latency for c_{ua} to X is given by:*

$$WCL^{Req}(c_{ua}) = \begin{cases} (2 + X_{ua} + Y_{ua}) \times SW & : if\ l \in \{A, B\} \\ 2 \times \left(\lceil \frac{|C^{CD}|}{T} \rceil \times (1 + R) \right. & \\ \qquad \left. + |C^{CD}| - 1 \right) \times SW & : if\ l \in \{C, D\} \end{cases}$$

Direct proof. Consider the following two cases. The first case is when $l \in \{A, B\}$, and the second case is when $l \in \{C, D\}$. Suppose c_{ua} such that $l \in \{A, B\}$ accesses the shared bus during its pre-allocated TDM slot. In the worst-case, c_{ua} attempts to broadcast a request immediately after the start of its TDM slot. Since requests must be broadcasted on the bus at the start of

the slot, c_{ua} can only successfully broadcast its request during its next pre-allocated TDM slot. Thus, c_{ua} must wait for the duration of its own slot whose start it just missed, and all other dedicated slots preceding its own next slot in the arbitration schedule (X_{ua}) resulting in a latency of $(1 + X_{ua})$ slots. In the worst-case, this slot is allocated for servicing write-back responses. As a consequence, c_{ua} must wait an additional $(1 + Y_{ua})$ for its next slot to broadcast its request. Therefore, in the worst-case, the broadcast request latency for c_{ua} is $(2 + X_{ua} + Y_{ua})$ slots. Suppose c_{ua} such that $l \in \{\mathsf{C}, \mathsf{D}\}$ accesses the shared bus. The worst-case scenario occurs when c_{ua} attempts to broadcast a request when all other $(|C^{\mathsf{CD}}| - 1)$ cores have pending requests. Hence, c_{ua} is blocked from successfully broadcasting its request while the $|C^{\mathsf{CD}}| - 1$ pending requests are serviced. These $|C^{\mathsf{CD}}| - 1$ cores can access the bus over multiple round-robin arbitration rounds, which we compute by $\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil$. Therefore, c_{ua} must wait for $(\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil)$ number of reserve slots, and $(\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil)$ number of R slots, which comprises of TDM slots of C^{AB} cores. After this delay, c_{ua} receives a slot to issue its request. However, in the worst-case, this slot may be used to service c_{ua}'s write-back response. Consequently, c_{ua} must wait an additional latency of $(\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1 + R) + |C^{\mathsf{CD}}| - 1)$ slots resulting in a total broadcast request latency of $2 \times \Big(\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1 + R) + |C^{\mathsf{CD}}| - 1 \Big)$. □

Theorem 7. *The worst-case response latency for c_{ua} to receive X is given by:*

$$WCL^{Resp}(c_{ua}) = WCL^{Req}(c_{ua}) + L^{acc}$$

Direct proof. Data responses are also sent from shared memory at the start of the receiving core's slot. As a result, the worst-case response latency is equal to the worst-case request latency and L^{acc}. We omit the proof of this theorem as it is similar to the proof of Theorem 6. □

Lemma 10. *The worst-case communication latency of c_{ua} where $l \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$ when data is communicated across criticality levels occurs when c_{ua} issues a read or write request to line X such that (1) α levels A and B cores broadcast write request to X before c_{ua}'s request is broadcasted, and (2) β levels C and D cores broadcast write requests to X before c_{ua}'s request is broadcasted.*

Direct proof. There are two cases to consider. The first case proves by contradiction that a scenario in which at least one A-D core broadcasts a read request instead of a write request to X does not result in the worst-case communication latency. The second case proves by contradiction that a scenario in which at least one A-D core broadcasts a write request after c_{ua} broadcasts its request to X does not result in the worst-case interference latency. We use Figure 6.5 to assist in the readability of the proof by contrasting these cases with the worst-case scenario for a 4-core MCS multi-core platform.

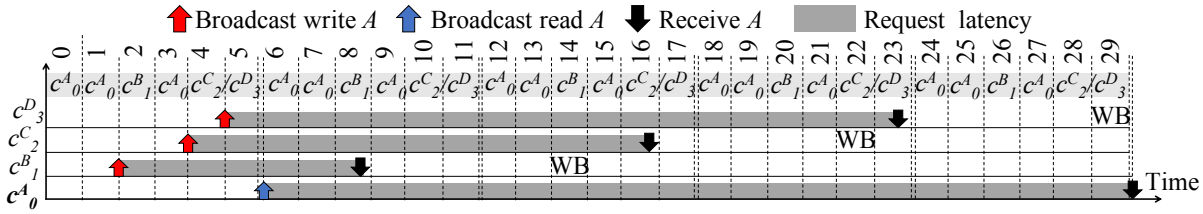


Figure 6.5: Worst-case instance for a 4-core system. c_{ua} is c_0^A.

Suppose $\exists\ c_i^l$ where $l \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$ that broadcasts a read request to X instead of a write request. Recall from Section 6.2 that a read request from one core does not require write-backs from other cores (SWMR invariant). As a result, c_{ua} will not experience any interference from c_i^l. Hence, c_{ua}'s data response will incur communication latency from $\alpha + \beta - 1$ cores, which is less than that from $\alpha + \beta$ cores. As an example, consider the following modification to the worst-case scenario depicted in Figure 6.5: c_2^C broadcasts a read request to X instead of a write request. c_2^C receives X in slot 16. Since c_2^C's request is a read, and does not modify X, c_3^D can receive X and complete its write request in slot 17. c_0^A waits for c_3^D to complete the write-back for X, and receives X in slot 24, which is less than the worst-case instance (30 slots).

Suppose $\exists\ c_i^l$ where $l \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$ that broadcasts a write request after c_{ua} broadcasts its request to X. As a result, c_i^l will receive X *after* c_{ua} receives X. Hence, c_{ua}'s data response will not incur any interference from c_i^l resulting in communication latency less than that when $|\alpha(l)|+|\beta(l)|$ cores broadcast write requests before c_{ua} broadcasts its request to X. As an example, consider the following modification to worst-case scenario depicted in Figure 6.5: c_1^B broadcasts its write request to X after c_0^A broadcasts its read request to X. Therefore, c_0^A receives X before c_1^B in slot 24, and does not incur the latency of c_1^B to do a write-back resulting in a lower WCL.

□

Lemma 11. *For c_{ua} where $l \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$, the maximum number of level A-B cores ($\alpha(l)$), and the maximum number of level C-D cores ($\beta(l)$) that can broadcast requests before c_{ua} broadcasts its request is given by:*

$$\alpha(l) = \begin{cases} |C^{AB} \setminus \{c_{ua}\}| & : if\ l \in \{\mathsf{A}, \mathsf{B}\} \\ |C^{AB}| & : if\ l \in \{\mathsf{C}, \mathsf{D}\} \end{cases}$$

$$\beta(l) = \begin{cases} min(|C^{CD}|, \lceil \frac{WCL^{Req}(c_{ua})}{P} \rceil \times (T-1) & \\ \qquad + max(0, (O - |C^E|))) & : if\ l \in \{\mathsf{A}, \mathsf{B}\} \\ |C^{CD} \setminus \{c_{ua}\}| & : if\ l \in \{\mathsf{C}, \mathsf{D}\} \end{cases}$$

where O is the total number of slack slots in $WCL^{Req}(c_{ua})$, and is computed as $O = WCL^{Req}(c_{ua}) - \lceil \frac{WCL^{Req}(c_{ua})}{P} \rceil \times T - \alpha$.

Direct proof. Suppose $l \in \{\mathsf{A}, \mathsf{B}\}$. From Theorem 6, WCL_{ua}^{Req} comprises of X_{ua} slots in which $|C^{AB}|-1$ cores can broadcast their requests to X. Hence, $|\alpha(l)| = |C^{AB} \setminus \{c_{ua}\}|$ when $l \in \{\mathsf{A}, \mathsf{B}\}$.

Suppose $l \in \{\mathsf{C}, \mathsf{D}\}$. From Theorem 6, WCL_{ua}^{Req} comprises of at least R slots in which $|C^{\mathsf{AB}}|$ cores can broadcast their requests to X Hence, $|\alpha(l)| = |C^{\mathsf{AB}}|$ when $l \in \{\mathsf{C}, \mathsf{D}\}$.

Suppose $l \in \{\mathsf{A}, \mathsf{B}\}$. WCL_{ua}^{Req} comprises of $\lceil \frac{WCL_{ua}^{\text{Req}}}{P} \rceil$ round-robin arbitration phases for C^{CD} cores to broadcast requests. Since the round-robin arbitration phase is of length T slots, and consists of one reserve slot where cores are allowed to complete pending requests but cannot broadcast requests, $\lceil \frac{WCL_{ua}^{\text{Req}}}{P} \rceil \times (T-1)$ level C-D cores can broadcast requests in WCL_{ua}^{Req}. Furthermore, we allow C^{CD} cores to broadcast requests in slack slots that are not utilized by C^E cores. The total number of slack slots is denoted as O. If $O - |C^E| > 0$, then there are slack slots that can be utilized by C^{CD} cores to broadcast requests. If $O - |C^E| \leq 0$, then there are enough C^E cores to utilize the slack slots. The maximum number of C^{CD} cores that can broadcast requests to X is bounded by $|C^{\mathsf{CD}}|$. Suppose $l \in \{\mathsf{C}, \mathsf{D}\}$. WCL_{ua}^{Req} comprises of $|C^{\mathsf{CD}}|$ slots where all remaining $|C^{\mathsf{CD}}| - 1$ receive a slot to broadcast requests. As a result, $|C^{\mathsf{CD}}| \setminus \{c_{ua}\}$ cores can broadcast requests before c_{ua} broadcasts its request. □

Theorem 8. *The worst-case communication latency incurred by c_{ua}'s request to X where $l \in \{\mathsf{A}, \mathsf{B}\}$ due to c_j^m's interfering write request to X is given by:*

$$WCL_{l,m}^{Comm} = \begin{cases} \lceil \frac{2}{s_j} \rceil \times P \times SW & : \text{if } m \in \{\mathsf{A}, \mathsf{B}\} \\ (2 \times \lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil) \times P \times SW & : \text{if } m \in \{\mathsf{C}, \mathsf{D}\} \end{cases}$$

Direct proof. c_j^m requires two allocated slots after broadcasting its request to receive the data for X, and write-back X. This is because after c_j^m broadcasts its request to X, c_j^m receives X from memory, and must write-back X in the next slot resulting in a latency of 2 allocated slots of c_j^m. Suppose c_j^m such that $m \in \{\mathsf{A}, \mathsf{B}\}$. Since c_j^m requires two slots to receive, update, and write-back X, the total communication latency due to c_j^m's request to X is $\lceil \frac{2}{s_j} \rceil \times P$ slots.

Suppose c_j^m such that $m \in \{\mathsf{C}, \mathsf{D}\}$. Since c_j^m is granted access to the bus using RR arbitration policy, c_j^m must wait $(\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil - 1) \times P$ slots to receive X. Hence, the total communication latency due to c_j^m's request to X is $(2 \times \lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil) \times P$ slots. □

Theorem 9. *The worst-case communication latency incurred by c_{ua}'s request to X where $l \in \{\mathsf{C}, \mathsf{D}\}$ due to c_j^m's interfering write request to X is given by:*

$$WCL_{l,m}^{Comm} = \begin{cases} 2 \times \Big(\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1+R) + \\ \qquad |C^{\mathsf{CD}}| \Big) \times SW & : \text{if } m \in \{\mathsf{A}, \mathsf{B}\} \\ 3 \times \Big(\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1+R) + \\ \qquad |C^{\mathsf{CD}}| \Big) \times SW & : \text{if } m \in \{\mathsf{C}, \mathsf{D}\} \end{cases}$$

Direct proof. Suppose c_j^m such that $m \in \{\mathsf{A}, \mathsf{B}\}$. Recall that a core requires two allocated slots to receive and modify X, and write-back X. As a result, c_j^m requires $\lceil \frac{2}{s_j} \rceil \times P$ slots to complete

these operations on X. Since R denotes the number of slots between two successive round-robin arbitration phases in the schedule, and $|C^{\mathsf{CD}}| > T$, $2 \times (\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1+R) + |C^{\mathsf{CD}}|) > \lceil \frac{2}{s_j} \rceil$. Hence, c_{ua} must wait $2 \times (\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1+R) + |C^{\mathsf{CD}}|)$ slots to receive X. Suppose c_j^m such that $m \in \{\mathsf{C}, \mathsf{D}\}$. When $|C^{\mathsf{CD}}| > T$, in the worst-case, c_j^m is allocated one slot in every round-robin arbitration phase. As a result, c_j^m requires $2 \times (\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1+R) + |C^{\mathsf{CD}}|)$ slots to receive and modify X, and write-back X. Since c_{ua} is also allocated one slot in every round-robin arbitration phase in the worst-case, c_{ua} must wait $3 \times (\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1+R) + |C^{\mathsf{CD}}|)$ slots to receive X. □

Theorem 10. *The total worst-case latency incurred by c_{ua}'s request to* X *where $l \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$ is given by:*

$$WCL^{Total}(c_{ua}) = \begin{matrix} WCL^{Req}(c_{ua}) + \sum_{i \in \alpha(l)+\beta(l)} WCL^{Comm}_{l,i} \\ +WCL^{Resp}(c_{ua}) \end{matrix}$$

Direct proof. The total WCL of c_{ua}'s request includes the worst-case request and data response latencies, and the worst-case communication latency due to interfering cores $|\alpha(l)| + |\beta(l)|$. □

6.6.3 Discussion

A key distinguishing feature between the WCL bounds of PMSI [74] and CARP is that WCL bounds in CARP are *independent* of the number of level E cores. On the other hand, WCL bounds in PMSI increase with level E cores as PMSI *elevates* level E cores to critical cores. Hence, CARP provides *tighter* WCL bounds for critical cores compared to PMSI.

The above analysis derives the *per-request* WCL bound to shared data that has *both* read and write permissions. On the other hand, the WCL bound of a request to shared data that has *only* read permissions is SW cycles as there are no coherence transitions for read-only shared data. Using static analysis tools, we envision the following process that utilizes the derived WCL bounds to compute the end-to-end WCET of a task. Static analysis tools can provide information on (1) the read/write patterns to a memory address and (2) the number of cores accessing a memory address. Based on the read/write patterns to a memory address, the appropriate WCL bound can be used, and the number of cores accessing a memory address can be used to substitute the parameters α and β derived previously in the analysis. Hence, the WCET of the task can be derived by applying this procedure to the memory addresses accessed by the task.

6.7 Methodology

We use gem5 [17] to evaluate CARP. gem5 is a micro-architectural simulator that models the memory subsystem and coherence protocol with high precision. CARP is simulated on a multi-core platform that comprises of 8 cores (c_0-c_7) running at 2GHz. Our simulated multi-core

Table 6.6: Hybrid arbitration policy parameters.

Parameter	Value
P	15 slots
X_0, X_1, X_2, X_3	11, 11, 13, 13 slots
Y_0, Y_1, Y_2, Y_3	0 slots
R	12 slots
T	3 slots
SW	50 cycles
$\lvert\alpha(\mathsf{A/B})\rvert$, $\lvert\alpha(\mathsf{C/D})\rvert$	3 cores, 4 cores
$\lvert\beta(\mathsf{A/B})\rvert$, $\lvert\beta(\mathsf{C/D})\rvert$	2 cores, 2 cores

platform does not run an OS. The cores implement in-order pipelines, and cores can have a single pending memory request. We allocate the following criticalities to the cores: c_0^A, c_1^A, c_2^B, c_3^B, c_4^C, c_5^C, c_6^D, c_7^E. Note that these allocations to cores are only done for empirical evaluation, a different mapping is also possible. Each core has a private L1 32KB 4-way instruction cache and 32KB 4-way data cache. The access latency to each private cache is set to 3 cycles. All cores share a 1MB set associative last-level cache (LLC). We configure the LLC such that all LLC accesses are hits in order to isolate and focus on the impact of maintaining cache coherence on the shared data access latencies. We set the LLC access latency to 50 cycles. Both the private L1 caches and shared LLC operate on cache line sizes of 64 bytes. The cores and the shared LLC are connected via a shared snooping bus that deploys an instance of the generalized arbitration policy described in Table 6.6.

We evaluate CARP against prior data communication mechanisms proposed for multi-core real-time and MCS platforms such as (1) *duplication* of communicated data [60, 79], (2) *cache bypassing* of communicated data [60, 61, 86], (3) *mapping* communicating tasks to the same core [25], and (4) the recently proposed *PMSI cache coherence protocol* [74]. For the PMSI cache coherence protocol, we elevate level E cores to critical cores as PMSI was designed for multi-core real-time systems where all cores are of the same criticality level [74]. We also evaluate CARP against MSI and MESI conventional cache coherence protocols [138]. Prior work showed that that deploying conventional coherence protocols on a predictable bus arbitration scheme can result in unbounded latencies for shared data accesses [74]. Hence, the conventional MSI and MESI protocols are executed on a snooping bus that does not deploy a predictable arbitration policy. We do not evaluate CARP against Pendulum [139] as it does not deal with tasks of varying criticality levels, and does not provide enough guidance on setting timer values.

Our evaluation uses synthetic benchmarks and SPLASH-2 [157], a multi-threaded benchmark suite. In the synthetic benchmarks (Synth1-Synth9), all cores except c_7^{E} perform the *same* sequence of operations (read/write) on shared data. c_7^{E} only performs read operations on shared data. As a result, these benchmarks stress the states and transitions of CARP. The synthetic

Table 6.7: Observed WCL for synthetic benchmarks.

Level	Analytical WCL (cycles)	Observed WCL (cycles)
A	6600	4348
B	6800	3701
C/D	11200	6699

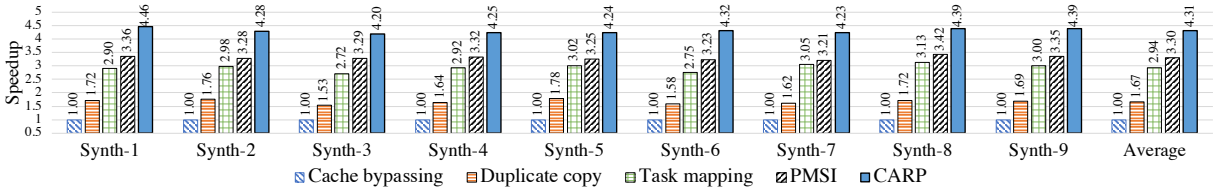


Figure 6.6: Performance of design choices and CARP on synthetic workloads.

benchmarks vary with each other based on the proportion of read and write operations. For these benchmarks, we run our simulation for 100,000 total memory operations across all cores. The SPLASH-2 benchmark suite consists of multi-threaded benchmarks derived from various domains such as graphics rendering, and scientific computation [157]. We use the SPLASH-2 benchmark suite due to a lack of available multi-threaded applications that operate on shared data, and are representative of those used in MCS. We run all SPLASH-2 benchmarks until completion. We used the SPLASH-2 benchmark to verify the data correctness of CARP, and observed that all benchmarks terminated with correct data output when executed using CARP.

6.8 Results

6.8.1 Synthetic workloads

Observed WCL. Table 6.7 shows the per-request observed total WCL across synthetic benchmarks deployed on CARP. CARP guarantees that the observed WCL are within the computed WCL bounds for critical cores across all benchmarks. In all benchmarks, the maximum observed request, data response, and communication latency components are within their respective analytical WCL bounds. We observe that the maximum request, and data response are equal to the respective analytical WCL bounds, and the maximum observed communication latencies vary across benchmarks as they are dependent on the memory activity on shared data. Benchmarks do not reach the maximum communication latency derived from the analysis because cores complete their requests earlier due to slack slots, and work conserving RR arbitration policies.

Average-case performance. Figure 6.6 shows the speedup in total execution time for the synthetic benchmarks using the design choices described earlier and CARP. We normalize the execution time to the cache bypassing data communication mechanism. Duplicating communicated

data and mapping tasks to the same core offer better performance over private cache bypassing as they allow communicated data to reside in the private caches. As a result, these techniques offer 1.67x and 2.94x performance speedup over cache bypassing. Note that duplicating communicated data increases the arbitration period as it requires level A-D cores to communicate updates to communicated data to the duplicate copies, and hence, does not perform as well as the task mapping technique. Mechanisms that use cache coherence (PMSI and CARP) offer better performance over task co-location technique as these mechanisms do not restrict task parallelism, and allow multiple copies of communicated data to reside in the private caches of all cores. CARP performs better than PMSI (30% on average) due to slack allocation for level E cores and the MCS arbitration schedule. Unlike PMSI, CARP is deployed on an arbitration policy that allocates different number of slots to cores based on their criticality levels. As a result, level A-D communicate more than one data in an arbitration period, which improves their communication throughput. CARP adds additional hardware logic such as looking up the PR buffer for aborting and retrying level E cores' requests, and identifying write-back candidates in the non-critical PWB buffer. We evaluated CARP with slot width set to 55 cycles to model the overhead of these lookups, and found that CARP still offered better performance improvements over PMSI (10% on average).

Performance slowdown relative to MSI and MESI. We evaluate the performance slowdown of CARP relative to the conventional MSI and MESI cache coherence protocols. Conventional coherence protocols (MSI and MESI) are designed for average-case performance, and are not designed to be predictable [49, 74]. These protocols are not deployed on a predictable bus arbitration mechanism. As a result, a core can broadcast its request as soon as the CC generates the request or immediately complete a write-back response on observing a remote write request. CARP exhibits an average performance slowdown of 73% and 66% compared to the MESI and MSI cache coherence protocols respectively. We find this slowdown reasonable for achieving predictability.

6.8.2 SPLASH-2 workloads

For SPLASH-2 workloads, we run CARP with level A-D cores and no level E cores. This is because, in SPLASH-2 workloads, *all* launched threads read and write on shared data structures such as locks and conditional barriers during execution in order to maintain benchmark correctness. We confirmed that the observed WCL bounds for level A-D cores are within the analytical bounds (not shown). Figure 6.7 shows the performance speedup of CARP compared to PMSI and the conventional MSI cache coherence protocol. The results are normalized to PMSI. We observe that CARP offers a performance improvement of 4% over PMSI, and shows an average slowdown of 60% compared to the conventional MSI protocol. In these benchmarks, thread barriers force all threads to converge at the barrier before making forward progress. These barriers

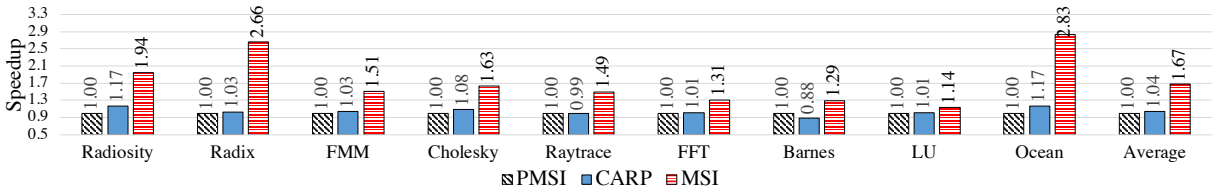


Figure 6.7: Performance of CARP on SPLASH-2.

are critical to maintain benchmark correctness. As a consequence, level A and B cores that may benefit from the additional allocated slots end up waiting for level C and D cores that have fewer allocated slots resulting in increased execution time.

6.9 Related works

There has been limited attention towards designing predictable data communication between tasks of different criticality levels in a multi-core MCS platform [14, 25]. Chisholm et al. [25] proposed a technique to allow data communication between levels A-C tasks in a multi-core MCS platform. Their technique applied mechanisms such as cache bypassing, mapping communicating tasks to one core, cache locking, and data duplication to enable task communication. As a result, their technique relies on OS and hardware support. In contrast, CARP is a hardware technique that does not restrict the usage of caches, does not require a particular mapping of tasks to cores, and does not duplicate communicated data. Becker et al. [14] proposed an alternative approach that constructed offline schedules of computation and memory phases of tasks such that contending data communication between multiple tasks were not scheduled at the same time. This approach relied on the availability of memory and compute details of the real-time tasks. CARP on the other hand, does not require information about the compute and memory behavior of the tasks.

Prior data communication techniques for multi-core real time systems used three main approaches: 1) disabled caching of communicated data in private caches [25, 61, 86], 2) replicated communication data [79], and 3) scheduled tasks that communicate data with each other on the same core through OS changes [23, 25, 52, 79]. These approaches provided predictable data sharing at the cost of reduced average-case performance. Alternatively, Pyka et al. [115] modified the application such that data communication on the same data were protected using software locks. The effect was that only one core performed data communication at any time instance. CARP on the other hand allows multiple cores to carry out their data communication simultaneously. Hassan et al. [74] recently described a set of guidelines for predictable data communication between real-time tasks using hardware cache coherence, and proposed PMSI, a predictable cache coherence protocol to that was built on these guidelines. CARP is also built using these

design guidelines, and includes additional features that enable CARP to be criticality-aware, and satisfy requirements specific to MCS. Sritharan et al. [139] recently proposed Pendulum, a time-based cache coherence protocol for dual-critical MCS systems. In this protocol, cores retain lines in their private caches for a duration of time period irrespective of the remote memory activity on the same lines, and the lines self-invalidate after the time period. The key novelty in Pendulum was that the time periods for a line were configured based on (1) the owner's criticality level and (2) the remote cores' criticality levels that issued memory requests to the same line. As a consequence, in Pendulum, memory activity of non-critical cores on shared data contributed to the WCL bounds of critical cores [139], which further requires an extension to the safety guidelines put forward in the ISO-26262 and AUTOSAR standards. On the other hand, CARP is designed to allow for data communication between critical and non-critical cores such that memory activity of non-critical cores do not contribute to the WCL bounds of critical cores.

Recently, Sensfelder et al. [133] provided a formal framework to model and analyze the latency interference in conventional bus based cache coherence protocols. We are motivated to apply similar models to CARP in order to formally verify the sources of interference due to data sharing in MCS, and reserve this exploration for future work.

6.10 Conclusion

As MCS platforms continue to integrate multiple complex tasks of varying criticality levels, enabling data communication between these tasks will be essential to realize added functionality and improve the overall responsiveness of the MCS. In this work, we present CARP, a criticality-aware hardware cache coherence protocol to realize predictable and high performance data communication between tasks executing on a multi-core MCS without violating the safety requirements of critical tasks. Our evaluation of CARP using synthetic and real-world benchmarks show that CARP guarantees the safety requirements of critical tasks, and improves average-case performance of MCS compared to prior techniques.

Chapter 7

End-to-End Predictable and High-Performance Real-Time Multi-Core Platforms

Designing real-time multi-core platforms that deliver end-to-end timing predictability and high average-case performance involves designing several pieces spanning across the software and hardware compute stack. In this chapter, we show how the research contributions in this thesis form a piece of this design exercise. We first describe the interaction between this thesis's research contributions and other timing compositional prior works to realize an end-to-end timing predictable and high-performance multi-core compute architecture. Next, we discuss at a high level how the timing analyses in this thesis can be incorporated into existing static timing analysis techniques to compute a task's total worst-case execution time (WCET).

Timing compositionality. Timing analysis of real-time multi-core platforms requires accounting for all possible sources of timing interference when computing a task's WCET. Prior approaches to timing analysis used a *non-compositional* timing analysis approach that treated the multi-core platform as a single *indivisible* unit. As a result, such timing analysis captured all possible timing dependencies caused by accesses to shared resources (large state space), which made the timing analysis extremely complex [59]. To deal with this complexity of the timing analysis, recent works on achieving predictability in multi-core platforms rely on *compositional timing analysis* that break away from the assumption that the multi-core platform is an indivisible unit [59]. The key insight of compositional timing analysis is that a multi-core platform can be decomposed into a set of *independent* components where the contribution of each component to the total WCET can be analyzed separately [59]. The total WCET of a task on the multi-core platform can be computed as a combination of the timing contributions. Prior works on designing predictable memory hierarchies [60, 117], shared memory buses [63, 66, 132], and

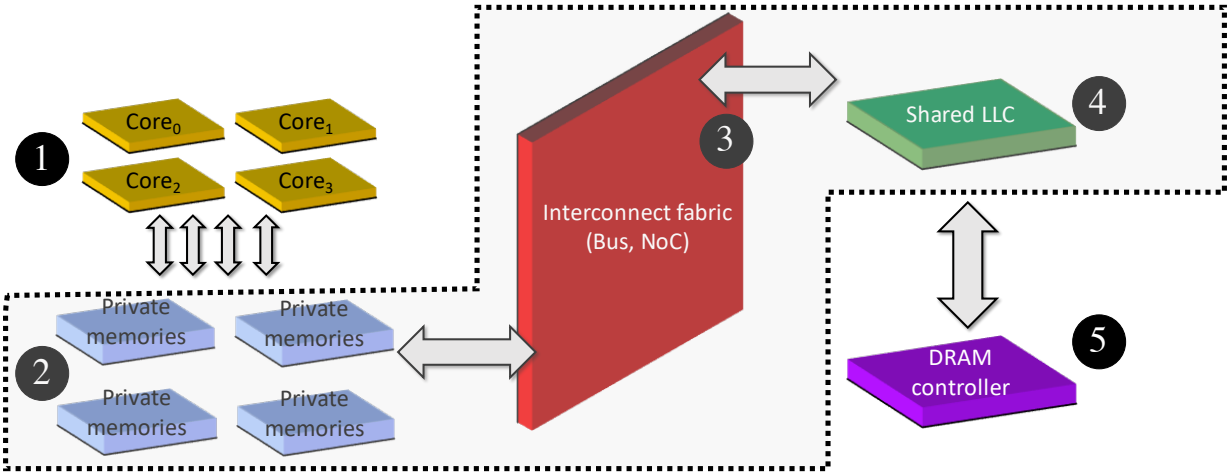


Figure 7.1: Hardware compute stack of a real-time multi-core platform. Research contributions in this thesis span across the highlighted layers.

predictable DRAM memory controllers [78, 159] relied on compositional timing analysis and derived component specific contributions to the WCET. In this chapter, we decompose a task's WCET on a multi-core compute platform into two components: (1) time spent in cores' compute pipeline circuitry ($WCL_{compute}$) and (2) time spent on memory accesses and the interconnects to access the shared memories (WCL_{memory}).

7.1 POTPOURRI: A (hypothetical) timing predictable and high-performance real-time multi-core platform

The research contributions of this thesis focus on achieving timing predictability and high-performance of one component of the compute stack, which is the shared data communication between multiple cores. Hence, this thesis adopts the *predictable computer architecture* design philosophy, which we described in Section 1.3.2 in Chapter 1. In this section, we show how our research contributions *fit* in a real-time multi-core compute stack that is fitted with components from other prior works that adopt the same predictable computer architecture design philosophy.

Figure 7.1 shows the hardware compute stack of a 4-core multi-core platform. For each layer of the hardware compute stack (cores, private memories, and shared memories) we list different designs that can be used. Table 7.1 describes one instance of POTPOURRI that chooses a specific prior work for a hardware compute layer along with our predictable cache coherence mechanisms.

Core architecture ❶. This level consists of the compute pipelines and control circuitry in the cores. A recent work by Hahn and Reineke [58] presented the SIC core, a provably timing

Compute layer	Design technique	Description	Contribution to WCET
Core ❶	SIC cores by Hahn and Reineke [58]	In-order cores free from timing anomalies	$WCL_{compute}$
Private memories ❷	Private split L1 caches with selfish LRU replacement [119]	Improved identification and analysis of cache hits and cache misses	WCL_{memory}
Shared interconnect fabric ❸	Shared bus with work-conserving TDM based arbitration [66]	Temporal isolation of shared memory accesses while improving average-case performance due to work-conserving nature	WCL_{memory}
Shared LLC ❹	Cache locking and cache partitioning support [25, 87]	Cache locking for frequently accessed shared data and cache partitioning for per-core private data	WCL_{memory}
Shared data communication ❷, ❸, ❹	PMESI* protocol from Chapter 4	Predictable and high-performance shared data communication mechanism	WCL_{memory}
Shared DRAM main-memory ❺	Conservative open-page policy DRAM memory controller [48]	Better average-case performance due to performance optimized page-policy while providing timing predictable	WCL_{memory}

Table 7.1: An instance of POTPOURRI using different related works that adopt predictable computer architecture design philosophy.

predictable pipelined processor core. A SIC core has an in-order pipeline and is designed to be free of timing anomalies [147]. Evaluation of the SIC core showed that it retains 93%-94% of the average-case performance of a conventional in-order core while enabling precise timing analysis. Hence, the four cores in Figure 7.1 can be designed as SIC cores. Other candidates for the cores include the ones used in the PRET [33, 91] and T-CREST [128] projects.

Private memories ❷. A core's private memories are exclusive memory components accessed by the tasks running on the core. These can include private caches such as the L1 instruction and data caches, unified L2 cache, and software programmable memories (scratchpad memories). Software programmable memories are memory components that are managed by the software application; the data contents of these memories are loaded and flushed by the software application. Hence, software programmable memories offer better predictability (tighter WCET) guarantees than hardware managed caches [123, 137, 143, 154]. However, hardware managed caches offer advantages such as better average-case performance and software application portability. As a result, prior research works have focused on improving the timing predictability of caches by designing better cache replacement policies [119], cache writing policies [15] (write-through vs write-back), and data placement techniques [131]. Given the complimentary benefits of caches and software programmable memories, we envision a core's private memory hierarchy to have both caches and software programmable memories.

Shared interconnect fabric ❸. The shared interconnect fabric is the interface between the private memory components (private caches, software programmable memory) and the shared memory components (shared LLC and shared main-memory). As discussed in Section 1.2 in Chapter 1, the shared interconnect fabric is a source of timing interference that renders the timing

analysis of real-time applications on multi-core platforms challenging. There is a large body of research focused on improving the timing predictability of this shared interconnect fabric, which can either be a shared bus for small core counts [27, 47, 63, 66, 67, 109] or a network-on-chip (NoC) for large core counts [46, 73, 83, 129, 130, 155].

Note that while the timing analysis of the predictable cache coherence mechanisms presented in this thesis assume a shared snooping bus, snooping bus cache coherence protocols can also be deployed on NoCs provided they *exhibit snooping bus behavior* [2]. As a result, the snooping predictable cache coherence protocols proposed in this thesis can also be deployed on timing predictable NoCs with appropriate extensions to support snooping bus behavior.

Shared memories ❹, ❺. The shared memories include the shared LLC and DRAM based main-memory. Similar to the shared interconnect fabric, the shared memories are sources of timing interference that complicate the timing analysis. For shared LLC, a number of research works have focused on designing techniques such as data bypassing, cache partitioning and locking and partitioning miss status handling registers (MSHRs) to improve their timing predictability [20,86,87,94,141,150,153]. The DRAM based main-memory is controlled by a DRAM memory controller. This memory controller is responsible for enforcing different policies of the DRAM operation such as data placement (address mappings), page policy (open-page vs close-page), and command scheduling (first-come first-serve first-ready FCFS, round-robin) [69]. These policies operate within the timing constraints placed by the DRAM. To improve the timing predictability of DRAM accesses, several works have focused on designing predictable DRAM controllers [4, 48, 56, 64, 78, 118].

7.2 Deriving WCET under predictable cache coherence

In this section, we present a high-level description on computing a task's WCET with predictable cache coherence enabled. The objective of this section is to not provide a complete analysis technique that integrates the timing analyses presented in this thesis with prior static timing analysis techniques and tools such as [54,68,120,140,156]. Rather, we discuss some key insights about the impact of predictable cache coherence on a task's WCET. These insights can ultimately lead to the integration of the timing analyses in this thesis into existing static analysis tools.

Recall that a task's WCET is decomposed of $WCL_{compute}$ and WCL_{memory} latency components among which predictable cache coherence affects the latter component. WCL_{memory} accounts for the worst-case memory access latency of all memory accesses issued by the task. One derivation of WCL_{memory} is to assume that each memory access generated by a task takes $WCL_{coherence}$ cycles, which is the worst-case latency of a memory access under predictable cache coherence; the total number of memory accesses generated by a task can be extracted through static analysis of the task. However, such a WCL_{memory} is grossly *pessimistic* for two

key reasons. First, such a derivation assumes *all* memory accesses are subjected to the worst-case scenarios under predictable cache coherence mechanisms. However, this need not be true as a task may generate different types of memory accesses such as uncacheable memory accesses, accesses to cacheable memory regions shared across multiple cores or accesses to cacheable memory regions private to a core. A cacheable memory access looks up the contents of the private cache levels and depending on whether the access is a cache hit or cache miss, looks up subsequent cache levels and finally the shared memory. An uncacheable memory access bypasses the cache hierarchy and directly accesses the shared main-memory. Different types of memory accesses have different WCL. For example, a private memory region is accessed by only one core. As a result, the worst-case scenarios under predictable cache coherence mechanisms derived in this thesis do not apply to accesses to private memory regions as multiple cores cannot access a memory region private to a core. Second, this derivation does not take into account that some accesses during the execution of the task may be cache hits, which have significantly lower WCL. Prior works on static cache analysis techniques such as [54, 68, 120, 140] used information about the cache organization (cache structure, replacement policy) and the task's control flow to precisely identify memory accesses that will result in cache hits (must-analysis) or cache misses (may-analysis). Such static cache analyses are key towards deriving tighter WCL_{memory} compared to a derivation that assumes all memory accesses are cache misses and are resolved by the shared main-memory. Note that existing static cache analyses are limited to memory accesses to memory regions that are private to a task; these techniques do not take into account changes to a core's cache state due to shared memory region accesses from other cores.

To this end, deriving a tight WCL_{memory} requires some *classification* of memory regions such that prior must-and-may cache analysis techniques and the timing analyses presented in this thesis can be appropriately applied. One such classification classifies a task's memory regions into *uncacheable*, *private*, and *shared* memory regions. We denote these regions as M_u, M_p and M_s sets where $M_u \cap M_p \cap M_s = \phi$. This classification can be done through static analysis of the task. Equation 7.1 computes WCL_{memory} by applying the appropriate latency analysis techniques based on this classification.

$$\begin{aligned} WCL_{memory} = \Sigma_{m \in M_u} WCL_{smem} \\ + \Sigma_{m \in M_p} \Big(\text{isCacheHit}(m, CI) \times WCL_{cHit} \\ + (1 - \text{isCacheHit}(m, CI)) \times WCL_{cMiss} \Big) \\ + \Sigma_{m \in M_s} WCL_{coherence} \end{aligned} \tag{7.1}$$

For memory accesses to uncacheable memory regions M_u, $\Sigma_{m \in M_u} WCL_{smem}$ is the total worst-case memory latency for such accesses. Since such accesses cannot be cached in the cache hierarchy, all accesses will access the shared main-memory; WCL_{smem} is the worst-case shared

main-memory (DRAM) access latency. For memory accesses to private memory regions M_p, the function isCacheHit(m, CI) takes as input the memory access address and information about the cache organization and replacement policy, which is denoted as CI and returns whether the access is a cache hit or a cache miss. We rely on prior works such as [54, 68, 120, 140] for the implementation of isCacheHit(m, CI). A cache hit takes WCL_{cHit} cycles and a cache miss takes WCL_{cMiss} cycles to complete ($WCL_{cHit} < WCL_{cMiss}$). Note that WCL_{cMiss} includes cache look-up and fill latency and the latency to access the shared main-memory WCL_{smem}. For memory accesses to shared memory regions M_s, $WCL_{coherence}$ is the worst-case latency under predictable cache coherence mechanisms derived in this thesis. For a task that makes accesses to different types of memory regions, clearly the computation of WCL_{memory} in Equation 7.1 is tighter than a derivation that assumes each memory access takes $WCL_{coherence}$ cycles.

The worst-case timing analyses in this thesis computes the WCL under predictable cache coherence for the *worst-case* access pattern, which is when multiple cores simultaneously modify the same shared data. However, a task may exhibit *different* memory access patterns that are different from the worst-case access pattern. Examples of such access patterns include read-only, migratory sharing, and producer-consumer access patterns. Such patterns have lower WCL compared to that under the worst-case access pattern. Hence, the combination of knowledge about a task's access pattern and deriving *access pattern specific WCLs* provides opportunities to further tighten the derivation of the task's WCET. In particular, by extracting information about the access pattern across cores to memory addresses and applying the appropriate WCL for that access pattern, we can further tighten the total worst-case memory latency of accesses to shared memory regions, which in turns tightens the total task's WCET.

Chapter 8

Conclusion and Future Works

As societal demands for more complex and integrated functionalities from real-time systems continue to rise, reconciling average-case performance and timing predictability will become a crucial design consideration for real-time compute platform manufacturers. This thesis advocates the design and adoption of *predictable hardware cache coherence mechanisms* for achieving high average-case performance and high timing predictability for shared data communication between tasks deployed on multiple cores. The research contributions in this thesis shed light on the timing and design intricacies associated with hardware cache coherence and propose deign guidelines, techniques, and tools to design predictable and high-performance hardware cache coherence mechanisms. The key contributions of this thesis include:

1. Bringing to attention different timing unpredictable scenarios that can arise when cores' can cache copies of shared data in their private memories in a multi-core platform deployed in hard real-time systems and MCS.

2. A design template in the form of design guidelines to design timing predictable hardware cache coherence mechanisms. We present several cache coherence mechanisms using this design template and validate their timing predictability and average-case performance benefits.

3. A formal treatment and understanding of the relationship between predictable hardware cache coherence mechanism design and their timing predictable and average-case performance trade-offs.

4. A technique to design predictable hardware cache coherence mechanisms that exhibit high timing predictability, which is on-par with state-of-the-art communication mechanisms, while exhibiting significant average-case performance speedup over state-of-the-art communication mechanisms.

5. A tool that automates the construction of predictable and high-performance cache coherence protocols.

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

1. **Designing low power and secure hardware cache coherence mechanisms:** The design trifecta for today's compute systems (real-time and conventional) include: *high-performance, low power and security*. This thesis is primarily concerned with achieving timing predictability and high-performance. However, low-power and security are important design goals for real-time systems given their deployment environment and safety-critical nature respectively.

   For example, low-power compute systems are crucial for automotive domains [89, 92]. Low-power compute systems do not stress the vehicles' electrical power supply, which in turn manifests in longer driving ranges. Given that data communication in compute platforms contributes to a significant portion of the total power consumption [19], there is merit in designing energy-efficient predictable hardware cache coherence mechanisms. Energy-efficient predictable hardware cache coherence mechanisms can reduce the power consumption footprint of data communication by limiting the number of cache lookups by a core and the number of interactions with the DRAM main-memory. For example, exploring the use of snoop filters to reduce the number of cache look-ups by a core on observing memory activity on the snooping bus is one way to reduce the power consumption from the cache coherence mechanism [100, 125].

   Recent work by Yao et al. [160] highlighted the existence of a timing side-channel in existing hardware cache coherence mechanisms that can be exploited to leak sensitive information stored in the compute platform. Such security breaches can translate to catastrophic consequences in a safety-critical real-time setting. Hence, it is important to couple the predictable hardware cache coherence mechanisms presented in this thesis with RTOS-level memory access and permission controls to prevent such security breaches [160].

2. **Multi-processor predictable hardware cache coherence mechanisms:** The slowdown of Moore's law and the emergence of data-intensive computations such as machine learning workloads have capped the performance benefits provided by traditional multi-core compute platforms [32, 34]. To counter the slowdown of Moore's law and accommodate emerging and dominant application designs, compute platforms are now integrating multiple *accelerators* in the form of GPUs, programmable FPGAs, and custom machine learning compute engines. For example, the MPSoC deployed in Tesla vehicles features a conventional multi-core compute complex, a GPU, a neural-network accelerator (NNA), and custom hardware for image signal processing and video encoding [145]. While the

research contributions presented in this thesis focus on achieving predictable cache coherence within a multi-core compute platform, cache coherence *extends* across these multiple processors [6, 110, 113, 169]. These heterogeneous processors on an MPSoC share the same main-memory (DRAM) and have private memory hierarchies. As a result, data communication between these processors must be handled in a coherent manner, which makes cache coherence a natural solution. For MPSoCs deployed in safety-critical real-time systems, handling this data communication in a timing predictable manner while keeping in mind different latency and throughput sensitivities of different processors is crucial.

References

[1] Advanced Micro Devices. AMD64 architecture programmer's manual volume 2: System programming, 2006.

[2] N. Agarwal, L. Peh, and N. 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.

[3] Niket Agarwal, Li-Shiuan Peh, and Niraj K Jha. In-network coherence filtering: Snoopy coherence without broadcasts. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 232–243, 2009.

[4] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: A predictable sdram memory controller. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '07, page 251–256, New York, NY, USA, 2007. Association for Computing Machinery.

[5] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert Ian Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (Proceedings)*. York, 2020.

[6] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. Spandex: A flexible interface for efficient heterogeneous coherence. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 261–274. IEEE, 2018.

[7] Sebastian Altmeyer, Robert I Davis, Leandro Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. A generic and compositional framework for multicore response time analysis. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pages 129–138, 2015.

[8] ARM, 2018.

[9] AUTOSAR. Autosar model constraints. volume 4.3.1, 2017.

[10] AUTOSAR. Overview of functional safety measures in autosar. volume 4.4.0, 2018.

[11] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, et al. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4):1–37, 2014.

[12] Thomas G. Baker. Lessons learned integrating cots into systems. In *Proceedings of the First International Conference on COTS-Based Software Systems*, ICCBSS '02, page 21–30, Berlin, Heidelberg, 2002. Springer-Verlag.

[13] A. Bansal, J. Singh, Y. Hao, J.Y. Wen, R. Mancuso, and M. Caccamo. Reconciling predictability and coherent caching. In *9th Mediterranean Conference on Embedded Computing (MECO 2020)*, pages 1–6, Budva, Montenegro, June 2020.

[14] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24, 2016.

[15] Pedro Benedicte, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. HWP: Hardware Support to Reconcile Cache Energy, Complexity, Performance and WCET Estimates in Multicore Real-Time Systems. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:22, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[16] Jingyi Bin, Sylvain Girbal, D Gracia Perez, Arnaud Grasset, and Alain Merigot. Studying co-running avionic real-time applications on multi-core COTS architectures. In *Embedded Real Time Software and Systems Conference*, 2014.

[17] 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. *ACM SIGARCH Computer Architecture News*, 2011.

[18] Vamsi Boppana, Sagheer Ahmad, Ilya Ganusov, Vinod Kathail, Vidya Rajagopalan, and Ralph Wittig. Ultrascale+ mpsoc and fpga families. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–37. IEEE, 2015.

[19] Shekhar Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, page 746–749, New York, NY, USA, 2007. Association for Computing Machinery.

[20] Eric Bost. Hardware support for robust partitioning in freescale qoriq multicore socs (p4080 and derivatives). *Freescale Semiconductor, Inc., Tech. Rep.*, 2013.

[21] Garo Bournoutian and Alex Orailoglu. Dynamic, multi-core cache coherence architecture for power-sensitive mobile processors. In *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 89–97. IEEE, 2011.

[22] Alan Burns and Robert I. Davis. A Survey of Research into Mixed Criticality Systems. *ACM Computing Surveys*, pages 82:1–82:37, 2018.

[23] J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *ECRTS*. IEEE, 2009.

[24] Francisco J Cazorla, Jaume Abella, Enrico Mezzetti, Carles Hernandez, Tullio Vardanega, and Guillem Bernat. Reconciling time predictability and performance in future computing systems. *IEEE Design & Test*, 35(2):48–56, 2018.

[25] M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, and F. D. Smith. Reconciling the Tension Between Hardware Isolation and Data Sharing in Mixed-Criticality, Multicore Systems. In *RTSS*. IEEE, 2016.

[26] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V Adve, Vikram S Adve, Nicholas P Carter, and Ching-Tsun Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 155–166. IEEE, 2011.

[27] B. Cilku, B. Frömel, and P. Puschner. A dual-layer bus arbiter for mixed-criticality systems with hypervisors. In *International Conference on Industrial Informatics (INDIN)*, pages 147–151. IEEE, July 2014.

[28] ARM Cortex. Cortex-A9 MPCore. *Technical Reference Manual*, 2009.

[29] D. Dasari, B. Akesson, V. Nélis, M. A. Awan, and S. M. Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 39–48, 2013.

[30] B. D. de Dinechin, R. Ayrignac, P. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, Sep. 2013.

[31] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 561–574, 2017.

[32] Chris Edwards. Moore's law: What comes next? *Commun. ACM*, 64(2):12–14, January 2021.

[33] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (pret) machine. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, page 264–265, New York, NY, USA, 2007. Association for Computing Machinery.

[34] L. Eeckhout. Is moore's law slowing down? what's next? *IEEE Micro*, 37(04):4–5, jul 2017.

[35] M. Elver, C. J. Banks, P. Jackson, and V. Nagarajan. Verc3: A library for explicit state synthesis of concurrent systems. In *DATE*, 2018.

[36] Marco Elver and Vijay Nagarajan. Tso-cc: Consistency directed cache coherence for tso. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 165–176. IEEE, 2014.

[37] Marco Elver and Vijay Nagarajan. Rc3: Consistency directed cache coherence for x86-64 with rc extensions. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 292–304. IEEE, 2015.

[38] Pak Eunji. Revisiting Resource Partitioning for Multi-core Chips: Integration of Shared Resource Partitioning on a Commercial RTOS. 2017.

[39] Federal Aviation Administration (FAA). Position Paper CAST-32A. 2016.

[40] Gabriel Fernandez, Jaume Abella, Eduardo Quiñones, Christine Rochange, Tullio Vardanega, and Francisco J Cazorla. Contention in multicore hardware shared resources: Understanding of the state of the art. In *14th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[41] Mikel Fernández, Roberto Gioiosa, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. Assessing the suitability of the ngmp multi-core processor in the space domain. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 175–184, 2012.

[42] International Organization for Standardization. ISO 26262, 2011.

[43] Takeshi Fukuda, Tasuku Ishigooka, and Fumio Narisawa. Multicore Migration Study in Automotive Powertrain Domain. 2017.

[44] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. Autosar–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62, page 5, 2009.

[45] Phani Kishore Gadepalli, Gregor Peach, Gabriel Parmer, Joseph Espy, and Zach Day. Chaos: a System for Criticality-Aware, Multi-core Coordination. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019.

[46] Tushar Garg, Saud Wasly, Rodolfo Pellizzoni, and Nachiket Kapre. Hoplitebuf: Network calculus-based design of fpga nocs with provably stall-free fifos. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(2):1–35, 2020.

[47] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Scheduling of Mixed-criticality Applications on Resource-sharing Multicore Systems. In *International Conference on Embedded Software (EMSOFT)*, pages 17:1–17:15. ACM, 2013.

[48] S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 525–530, 2013.

[49] G. Gracioli and A. A. Fröhlich. On the influence of shared memory contention in real-time multicore applications. In *Brazilian Symposium on Computing Systems Engineering*, pages 25–30, Nov 2014.

[50] G. Gracioli and A. A. Fröhlich. Two-phase colour-aware multicore real-time scheduler. *IET Computers Digital Techniques*, 2017.

[51] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Computing Surveys*, 2015.

[52] Giovani Gracioli and Antônio Augusto Fröhlich. On the Design and Evaluation of a Real-Time Operating System for Cache-Coherent Multicore Architectures. *ACM SIGOPS Operating Systems Review*, 2016.

[53] Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[54] Daniel Grund and Jan Reineke. Toward precise plru cache analysis. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[55] 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*, pages 22–31. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, March 2011.

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

[57] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, page 413–422, New York, NY, USA, 2009. Association for Computing Machinery.

[58] S. Hahn and J. Reineke. Design and analysis of sic: A provably timing-predictable pipelined processor core. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 469–481, 2018.

[59] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: Definition and challenges. *ACM SIGBED Review*, 2015.

[60] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication Centric Design in Complex Automotive Embedded Systems. In *ECRTS*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[61] D. Hardy, T. Piquet, and I. Puaut. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *RTSS*. IEEE, 2009.

[62] M. Hassan. Heterogeneous MPSoCs for Mixed-Criticality Systems: Challenges and Opportunities. *IEEE Design Test*, pages 47–55, 2018.

[63] M. Hassan and H. Patel. Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems. In *RTAS*. IEEE, 2016.

[64] M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *RTAS*. IEEE, 2015.

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

[66] F. Hebbache, M. Jan, F. Brandner, and L. Pautet. Shedding the shackles of time-division multiplexing. In *Real-Time Systems Symposium (RTSS)*, pages 456–468. IEEE, Dec 2018.

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

[68] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–212. IEEE, 2011.

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

[70] Natalie D Enright Jerger. *Chip multiprocessor coherence and interconnect system design*. PhD thesis, Citeseer, 2008.

[71] Natalie D Enright Jerger, Li-Shiuan Peh, and Mikko H Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 35–46. IEEE, 2008.

[72] Matthias Jung, Sally A. McKee, Chirag Sudarshan, Christoph Dropmann, Christian Weis, and Norbert Wehn. Driving into the memory wall: The role of memory for advanced driver assistance systems and autonomous driving. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, pages 377–386, New York, NY, USA, 2018. ACM.

[73] Evangelia Kasapaki and Jens Sparsø. Argo: A time-elastic time-division-multiplexed noc using asynchronous routers. In *2014 20th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 45–52. IEEE, 2014.

[74] A. M. Kaushik, M. Hassan, and H. Patel. Designing predictable cache coherence protocols for multi-core real-time systems. *IEEE Transactions on Computers*, pages 1–1, 2020.

[75] Anirudh Mohan Kaushik and Hiren Patel. Automated synthesis of predictable and high-performance cache coherence protocols. In *Design, Automation, and Test in Europe Conference (DATE)*, 2021.

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

[77] Anirudh Mohan Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren Patel. CARP: A Data Communication Mechanism for Multi-Core Mixed-Criticality Systems. In *Real-Time Systems Symposium (RTSS)*. IEEE, 2019.

[78] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh. A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 317–326. IEEE, April 2015.

[79] N. Kim, M. Chisholm, N. Otterness, J. H. Anderson, and F. D. Smith. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 223–234. IEEE, 2017.

[80] Namhoon Kim, Bryan C Ward, Micaiah Chisholm, James H Anderson, and F Donelson Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real-Time Systems*, 53(5):709–759, 2017.

[81] Rakesh Komuravelli, Sarita V Adve, and Ching-Tsun Chou. Revisiting the complexity of hardware cache coherence and some implications. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–22, 2014.

[82] Rakesh Komuravelli, Matthew D Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V Adve, and Vikram S Adve. Stash: Have your scratchpad and cache it too. *ACM SIGARCH Computer Architecture News*, 43(3S):707–719, 2015.

[83] Adam Kostrzewa, Selma Saidi, and Rolf Ernst. Slack-based Resource Arbitration for Real-time Networks-on-chip. In *Design, Automation & Test in Europe (DATE)*, pages 1012–1017. EDA Consortium, 2016.

[84] A. Kritikakou, C. Pagetti, O. Baldellon, M. Roy, and C. Rochange. Run-time control to increase task parallelism in mixed-critical systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 119–128. IEEE, July 2014.

[85] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. A communication-centric approach for designing flexible dnn accelerators. *IEEE Micro*, 38(6):25–35, 2018.

[86] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures. In *RTNS*, 2010.

[87] Benjamin Lesage, Isabelle Puaut, and André Seznec. PRETI: Partitioned real-time shared cache for mixed-criticality real-time systems. In *RTNS*. ACM, 2012.

[88] 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.

[89] 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*, ASPLOS '18, page 751–766, New York, NY, USA, 2018. Association for Computing Machinery.

[90] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1973.

[91] Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A Lee. A pret microarchitecture implementation with repeatable timing and competitive performance. In *2012 IEEE 30th international conference on computer design (ICCD)*, pages 87–93. IEEE, 2012.

[92] Shaoshan Liu, Jie Tang, Zhe Zhang, and Jean-Luc Gaudiot. Caad: Computer architecture for autonomous driving. *arXiv preprint arXiv:1702.01894*, 2017.

[93] Pejman Lotfi-Kamran, Michael Ferdman, Daniel Crisan, and Babak Faisafi. Turbotag: lookup filtering to reduce coherence directory power. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 377–382. IEEE, 2010.

[94] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, 2013.

[95] Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. Wcet (m) estimation in multi-core systems using single core equivalence. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 174–183. IEEE, 2015.

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

[97] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 2012.

[98] Lucia G Menezo, Valentin Puente, and Jose-Angel Gregorio. Flask coherence: A morphable hybrid coherence protocol to balance energy, performance and scalability. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 198–209. IEEE, 2015.

[99] Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, and John A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *International Conference on Computer and Information Technology (CIT)*, pages 1864–1871, Washington, DC, USA, 2010. IEEE Computer Society.

[100] Andreas Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 234–245. IEEE, 2005.

[101] Andreas Moshovos, Gokhan Memik, Babak Falsafi, and Alok Choudhary. Jetty: Filtering snoops for reduced energy consumption in smp servers. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 85–96. IEEE, 2001.

[102] Ann Steffora Mutschler. Car industry changing under the hood. 2021.

[103] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *European Dependable Computing Conference*. IEEE, 2012.

[104] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 109–118, 2014.

[105] NVIDIA. JETSON TK1: Unlock the power of the GPU for embedded systems applications, 2016.

[106] NXP. QorlQ® T4240, T4160 and T4080 Multicore Processors, 2018.

[107] N. Oswald, V. Nagarajan, and D. J. Sorin. ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications. In *ISCA*, 2018.

[108] N Oswald, V Nagarajan, and D J Sorin. Hieragen: Automated generation of concurrent, hierarchical cache coherence protocols. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 888–899, 2020.

[109] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*. ACM, 2009.

[110] Songwen Pei, Myoung-Seo Kim, Jean-Luc Gaudiot, and Naixue Xiong. Fusion coherence: scalable cache coherence for heterogeneous kilo-core system. In *Advanced Computer Architecture*, pages 1–15. Springer, 2014.

[111] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, 2011.

[112] Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella Ferrer, Francisco Javier Cazorla Almeida, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-core devices for safety-critical systems: A survey. *ACM Computing Surveys*, 53(4), 2020.

[113] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 457–467. IEEE, 2013.

[114] Roger Pujol, Hamid Tabani, Jaume Abella, Mohamed Hassan, and Francisco J. Cazorla. Empirical Evidence for MPSoCs in Critical Systems: The Case of NXP's T2080 Cache Coherence. In *IEEE Design Automation and Test in Europe (DATE)*, pages 1–4, December 2020.

[115] A. Pyka, M. Rohde, and S. Uhrig. Extended performance analysis of the time predictable on-demand coherent data cache for multi- and many-core systems. In *SAMOS*. IEEE, 2014.

[116] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4), January 2012.

[117] Syed Aftab Rashid, Geoffrey Nelissen, Sebastian Altmeyer, Robert I Davis, and Eduardo Tovar. Integrated analysis of cache related preemption delays and cache persistence reload overheads. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 188–198. IEEE, 2017.

[118] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *CODES+ISSS*. IEEE, 2011.

[119] Jan Reineke, Sebastian Altmeyer, Daniel Grund, Sebastian Hahn, and Claire Maiza. Selfish-lru: Preemption-aware caching for predictability and performance. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 135–144. IEEE, 2014.

[120] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[121] Alberto Ros and Alexandra Jimborean. A dual-consistency cache coherence protocol. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1119–1128. IEEE, 2015.

[122] Alberto Ros and Stefanos Kaxiras. Complexity-effective multicore coherence. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 241–252, 2012.

[123] Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut. Hiding communication delays in contention-free execution for spm-based multi-core architectures.

In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[124] RTCA Inc. Software Considerations in Airborne Systems and Equipment Certification. 1992.

[125] Valentina Salapura, Matthias Blumrich, and Alan Gara. Design and implementation of the blue gene/p snoop filter. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 5–14. IEEE, 2008.

[126] C. E. Salloum, M. Elshuber, O. Höftberger, H. Isakovic, and A. Wasicek. The ACROSS MPSoC – A New Generation of Multi-core Processors Designed for Safety-Critical Embedded Systems. In *Euromicro Conference on Digital System Design*, pages 105–113. IEEE, Sept 2012.

[127] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst. System level performance analysis for real-time automotive multicore and network architectures. *IEEE TCAD*, 2009.

[128] 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.

[129] Martin Schoeberl, David Vh Chong, Wolfgang Puffitsch, and Jens Sparsø. A time-predictable memory network-on-chip. In *14th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[130] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, 2018.

[131] Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. Towards time-predictable data caches for chip-multiprocessors. In *Software Technologies for Embedded and Ubiquitous Systems*. Springer Berlin Heidelberg, 2009.

[132] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing analysis for tdma arbitration in resource sharing systems. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 215–224. IEEE, 2010.

[133] Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. Modeling Cache Coherence to Expose Interference. In *ECRTS*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[134] Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. Modeling Cache Coherence to Expose Interference. In *Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[135] Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. On How to Identify Cache Coherence: Case of the NXP QorIQ T4240. In *ECRTS*, 2020.

[136] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. Cache coherence for gpu architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 578–590. IEEE, 2013.

[137] Muhammad Refaat Soliman and Rodolfo Pellizzoni. Wcet-driven dynamic data scratchpad management with compiler-directed prefetching. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[138] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 2011.

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

[140] Gregory Stock, Sebastian Hahn, and Jan Reineke. Cache persistence analysis: Finally exact. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 481–494. IEEE, 2019.

[141] Vivy Suhendra and Tulika Mitra. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-cores. In *DAC*. ACM/IEEE, 2008.

[142] Abdulaziz Tabbakh, Xuehai Qian, and Murali Annavaram. G-tsc: Timestamp based coherence for gpus. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 403–415. IEEE, 2018.

[143] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric os for multi-core embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, 2016.

[144] C. Takahashi, S. Shibahara, K. Fukuoka, J. Matsushima, Y. Kitaji, Y. Shimazaki, H. Hara, and T. Irita. 4.5 a 16nm finfet heterogeneous nona-core soc complying with iso26262 asil-b: Achieving 107 random hardware failures per hour reliability. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 80–81, 2016.

[145] E. Talpes, D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti, and G. S. Sachdev. Compute solution for tesla's full self-driving computer. *IEEE Micro*, 40(02):25–35, mar 2020.

[146] ARM Technology. Arm expects vehicle compute performance to increase 100x in next decade, 2015.

[147] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[148] V. Tran, Dar-Biau Liu, and B. Hummel. Component-based systems development: challenges and lessons learned. In *Proceedings Eighth IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering*, pages 452–462, 1997.

[149] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. Transit: Specifying protocols with concolic snippets. In *PLDI*, 2013.

[150] 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. IEEE, 2016.

[151] Dana Vantrease, Mikko H Lipasti, and Nathan Binkert. Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 132–143. IEEE, 2011.

[152] S. Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Real-Time Systems Symposium (RTSS)*, pages 239–243. IEEE, Dec 2007.

[153] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Making Shared Caches More Predictable on Multicore Platforms. In *ECRTS*. IEEE, 2013.

[154] S. Wasly and R. Pellizzoni. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 183–192, 2013.

[155] 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. IEEE, 2017.

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

[157] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*. IEEE, 1995.

[158] Chun Wah Wallace Wu, Deepak Kumar, Borzoo Bonakdarpour, and Sebastian Fischmeister. Reducing monitoring overhead by integrating event- and time-triggered techniques. In *Runtime Verification*, pages 304–321. Springer Berlin Heidelberg, 2013.

[159] Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst Case Analysis of DRAM Latency in Multi-requestor Systems. In *RTSS*. IEEE, 2013.

[160] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179. IEEE, 2018.

[161] G. Yao, H. Yun, Z. P. Wu, R. Pellizzoni, M. Caccamo, and L. Sha. Schedulability analysis for memory bandwidth regulated multicore real-time systems. *IEEE Transactions on Computers*, 65(2):601–614, 2016.

[162] Yuan Yao, Guanhua Wang, Zhiguo Ge, Tulika Mitra, Wenzhi Chen, and Naxin Zhang. Efficient timestamp-based cache coherence protocol for many-core architectures. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–13, 2016.

[163] Xiangyao Yu and Srinivas Devadas. Tardis: Time traveling coherence algorithm for distributed shared memory. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 227–240. IEEE, 2015.

[164] Xiangyao Yu, Hongzhe Liu, Ethan Zou, and Srinivas Devadas. Tardis 2.0: Optimized time traveling coherence for relaxed consistency models. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 261–274. IEEE, 2016.

[165] H. Yun, R. Pellizzon, and P. K. Valsan. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *ECRTS*. IEEE, 2015.

[166] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 299–308. IEEE Computer Society, 2012.

[167] Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K Qureshi, and Andreas Moshovos. A tagless coherence directory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–434, 2009.

[168] Guowei Zhang, Webb Horn, and Daniel Sanchez. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 13–25, 2015.

[169] Hongzhou Zhao, Arrvindh Shriraman, Snehasish Kumar, and Sandhya Dwarkadas. Protozoa: Adaptive granularity cache coherence. *ACM SIGARCH Computer Architecture News*, 41(3):547–558, 2013.

[170] X. Zheng, C. Julien, R. Podorozhny, and F. Cassez. BraceAssertion: Runtime Verification of Cyber-Physical Systems. In *International Conference on Mobile Ad Hoc and Sensor Systems*, pages 298–306. IEEE, Oct 2015.

[171] X. Zheng, C. Julien, R. Podorozhny, F. Cassez, and T. Rakotoarivelo. Efficient and Scalable Runtime Monitoring for Cyber–Physical System. *IEEE Systems Journal*, pages 1667–1678, June 2018.

[172] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Tbd: Benchmarking and analyzing deep neural network training. *arXiv preprint arXiv:1803.06905*, 2018.

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