Predictable Cache Coherence Protocols for Mixed-Time-Criticality Multi-core Systems

by

Nivedita Sritharan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2017

© Nivedita Sritharan 2017

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

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

Statement of Contributions

In Chapters 6, 7, 8, Hassan, M., Kaushik, A. M. and Patel, H., aided in formalising the idea of HourGlass, provided inputs on the architectural details of the protocol and timing analysis. Hassan, M. aided in writing the analysis for HourGlass(H-DD-WC-0). Kaushik, A. M., helped in Section 7.1 of Chapter 7.

Abstract

Modern real-time systems consist of a combination of hard real-time, firm real-time and soft real-time tasks. Hard real-time (HRT) tasks mandate strict timing requirements by requiring that a static timing analysis can be performed to compute a worst-case latency (WCL) bound. Firm real-time (FRT) and soft real-time (SRT) tasks, on the other hand, do not impose such stringent requirements. Instead, they tolerate infrequent violations of deadlines in favour of improved average-case performance. When deploying such a system on a multi-core platform, the hardware resources such as the main memory, caches and shared bus are shared between the tasks. This results in interference by FRT or SRT tasks on HRT tasks, which complicates the timing analysis for HRT tasks, and potentially yields unbounded WCL. This thesis presents a time-based cache coherence protocol, HourGlass, to predictably share data in a multi-core system across different criticality tasks. HourGlass is derived from the conventional Modified Shared Invalid (MSI) cache coherence protocol, and it is equipped with a timer mechanism that allows the cores to hold a valid copy of data in its private cache for certain duration. HourGlass is designed to ensure WCL bounds for HRT tasks, and it also provides performance improvements for FRT and SRT tasks. Such a coherence protocol encourages a trade-off between the WCL bounds for hard real-time tasks, and performance offered to firm or soft real-time tasks with the help of timer mechanisms. HourGlass was prototyped in gem5, a micro-architectural simulator, and evaluated with multi-threaded benchmarks.

Acknowledgements

I express my deep gratitude to my supervisor, Professor Hiren Patel, for his patience, encouragement, and training during my graduate studies. This thesis would not be possible without his support and guidance. I thank my readers for reviewing this thesis and providing valuable feedback: Professor Rodolfo Pellizzoni and Professor Nachiket Kapre.

I thank my colleagues in the Computer Architecture and Embedded Systems Research (CAESR) group at University of Waterloo for the valuable discussions, guidance and support: Yunling Cui, Mohammed Hassan, Anirudh M. Kaushik, and Paulos Tegegn.

I thank my parents for their endless support and encouragement throughout my life. I am grateful to them for always believing in me and helping me in fulfilling my dreams. I am thankful to my grandparents, who supported me in all my decisions and showered me with blessings and love.

I also thank my brother, Nikhil, without whose constant encouragement I would not have pursued my graduate studies. I am also grateful to my family, who were always there for me when I was homesick: Renuka, Kemburaj, Anu, Prabakaran, Kunal, Akkilaa, Vishal, Janani, Ajit, Krithika.

Special thanks to my wonderful friends here at Waterloo, who provided great moral support and made my stay here more lively and enjoyable: Kritika, Arshee, Jacqueline, Meghana, Namrah and Rabeeah.

Finally, I thank the Almighty for giving me the strength to complete my graduate studies.

Dedication

This thesis is dedicated to my parents, Radhika Sritharan and V. Sritharan, who have supported and encouraged me throughout my life. This thesis is possible only because of their constant love and sacrifices.

Table of Contents

List of Tables x

List of Figures xi

1 Introduction 1

2 Related Work 4

2.1 Real-time systems 4

2.2 Mixed time-criticality systems 5

2.3 Bus arbitration 6

2.4 Time-based coherence protocol 7

3 Background 9

3.1 Cache coherence 9

3.1.1 Coherence protocol 11

3.2 Mixed time-criticality multi-core systems 13

3.2.1 Arbitration schemes 14

4 System Model 16

5 Bus Arbitration schemes **20**

5.1 Dedicated slots for all cores (ALL-DD) 21

5.2 Dedicated slots for HRT cores with Non work conserving arbitration (H-DD-NWC) 22

5.3 Dedicated slots for HRT cores with Work-conserving arbitration (H-DD-WC) 23

6 Use of Criticality-aware Bus Arbitration and Timers - an Illustration **25**

7 HourGlass **30**

7.1 Architectural Modifications 30

7.1.1 Architectural modifications to shared bus 31

7.1.2 Architectural modifications to cache controllers 33

7.1.3 Hardware overhead 35

7.2 Cache Coherence Protocol Modifications 35

7.2.1 Modifications for Criticality awareness 37

7.2.2 Support for Timers 39

7.2.3 Illustrative Examples 40

8 Timing Analysis **47**

8.1 Timing Analysis for HourGlass(H-DD-NWC) 49

8.1.1 Bound for HRT cores 50

8.1.2 Bound for FRT cores 60

8.2 Timing Analysis for HourGlass(H-DD-WC) 71

8.2.1 Bound for HRT cores 71

8.2.2 Bound for FRT cores 78

9 Evaluation **86**

9.1 Data correctness and protocol verification 87

9.2 Bounding memory access latencies 87

9.3 Comparison of per request WCL bounds of HourGlass with other approaches 89
9.4 Comparison with other approaches 91
9.5 Effect of timers on the performance of {FRT,SRT} cores 95
9.5.1 Effect on HourGlass(H-DD-NWC) 97
9.5.2 Effect on HourGlass(H-DD-WC) 98
9.5.3 Effect on HourGlass(H-DD-WC-0) 98
9.6 Scalability 99
9.6.1 Effect on HourGlass(H-DD-NWC) 99
9.6.2 Effect on HourGlass(H-DD-WC) 100
9.6.3 Effect on HourGlass(H-DD-WC-0) 101

10 Conclusion and Future Work **103**

References **105**

List of Tables

7.1 State transitions at the private cache 36
7.2 Different state transitions based on core criticality and issuance of requests in *slack* or *dedicated* slots 38
7.3 State transitions at the shared memory 38

A1 HourGlass configurations. 89

List of Figures

3.1 Cache Coherence Protocol. 10
3.2 MSI Cache Coherence Protocol. 12
3.3 Cache Coherence Protocol with states. 13
3.4 Mixed time-criticality multi-core system. 14

4.1 Mixed time-criticality multi-core system. 17

5.1 Bus arbitration schemes. 21

6.1 PMSI cache coherence protocol with fixed priority arbitration. 26
6.2 Proposed solution with timers for SRT cores. 28
6.3 Proposed solution with timers for HRT cores. 29

7.1 Architectural Modifications. 31
7.2 Multiple pending requests from HRT and SRT cores. 41
7.3 Multiple sharers. 43
7.4 Multiple pending requests in HourGlass(H-DD-WC). 44

8.1 Different latency components per request of a core. 48
8.2 Illustration for the latency due to data transfer at requesting core slot. . . 49
8.3 Critical Instance for H-DD-NWC HRT core - Read-Write Unshared. 52
8.4 Critical Instance for H-DD-NWC HRT core - Read-Write Shared. 56
8.5 Critical Instance for H-DD-NWC FRT core - Read-Write Unshared. 63

8.6 Critical Instance for H-DD-NWC FRT core - Read-Write Shared. 67
8.7 Critical Instance for H-DD-WC-0 HRT core - Read-Write Shared. 72
8.8 Critical Instance for H-DD-WC FRT core - Read-Write Shared. 80

9.1 Observed WCL components for HourGlass(H-DD-NWC). 87
9.2 Observed WCL components for HourGlass(H-DD-WC). 88
9.3 Observed WCL components for HourGlass(H-DD-WC-0). 88
9.4 Observed per request WCL of HRT cores for all real-time approaches. . . . 90
9.5 Total execution time slowdown compared to MESI protocol. 91
9.6 Speedup compared to PMSI protocol - Synchrobench. 93
9.7 Speedup compared to PMSI protocol - AutoBench 2.0. 94
9.8 Speedup compared to PMSI protocol - Synthetic benchmarks. 94
9.9 Effect of timers in HourGlass(H-DD-NWC). 96
9.10 Effect of timers in HourGlass(H-DD-WC). 97
9.11 Effect of timers in HourGlass(H-DD-WC-0). 98
9.12 Effect of scalability in HourGlass(H-DD-NWC). 99
9.13 Effect of scalability in HourGlass(H-DD-WC). 101
9.14 Effect of scalability in HourGlass(H-DD-WC-0). 102

Chapter 1

Introduction

Modern real-time systems consist of a combination of tasks that may be of different criticality levels. These tasks may further share the underlying hardware resources when deployed on a computing platform. For certification, it is imperative to guarantee that tasks deemed to be of high criticality such as hard real-time tasks, never exceed their temporal requirements. Naturally, exceeding the temporal requirements for such hard real-time tasks (HRT) may render the system in a state with the potential for catastrophic consequences. However, modern real-time systems also consist of tasks that do not require such strict temporal requirements, but instead, those may only require best-effort service. Classical examples of such tasks include firm real-time tasks (FRT) and soft real-time tasks (SRT). A real-time system with a combination of HRT, FRT and SRT tasks is what we refer to as a mixed time-criticality system. Example domains include avionics and automotive (DO 178C and ISO 26262) [8], which have further extended the MCS model to multiple criticality levels. For instance, the Anti-lock Braking System (ABS) of the automotive is required to respond within a certain time duration, otherwise leading to life-threatening consequences (hard real-time). The automotive navigation system is required to provide results within a certain time duration, otherwise the result is invalid (firm real-time). However, they do not lead to life-threatening consequences like HRT tasks. The infotainment system, on the other hand, only requires best-effort service for the user and do not require any timing requirements (soft real-time).

As demands for more functionality and better performance from mixed time-criticality systems increase, there is noticeable interest in leveraging multi-core platforms to deploy such applications (Freescale P4080, [3]). The use of multi-core platforms reduces hardware cost, and it also offers true parallelism that applications can exploit for better performance. However, a primary concern in mixed time-criticality multi-core systems is the interference

caused between HRT, FRT and SRT tasks that share the hardware resources. This is because the FRT and SRT tasks, that do not have strict timing requirements, interfere with the HRT tasks. The contributing interference may originate from one or many of the hardware components such as the main memory, last-level caches, and shared buses. This interference can either cause the HRT tasks to miss their timing guarantees or the FRT and SRT tasks to contribute to the worst-case execution time (WCET) of HRT tasks. This significantly complicates the WCET analysis for HRT tasks because a static analysis of the HRT tasks must incorporate the effect of the FRT and SRT tasks' execution. Consequently, there is considerable interest in devising strategies to mitigate the impact of the interference such that HRT tasks are guaranteed to meet their requirements.

Multi-core platforms also introduce additional sources of interference. One such interference occurs when multiple tasks deployed on different cores access data that is temporarily stored in the private cache of a core. Since multiple threads may update the shared data, ensuring that any read of the shared data receives the most up-to-date write to the shared data is essential for correct execution. This involves transferring the data from one private cache, through the bus, to the requesting core's cache. Since the worst-case latency (WCL) to access data is a component of the task's WCET, predictably managing shared data across multiple tasks deployed on different cores remains a challenging problem [8].

General purpose micro-architectures use cache coherence in such scenarios. Cache coherence is a mechanism that maintains data coherently across the private caches of the cores to ensure correct and high performance execution of the application. However, conventional cache coherence protocols do not manage predictable transfer of shared data across the cores. Hence, prior work has focused on disabling caches to predictably share data across multiple cores [18, 24], or only caching non-shared data [10]. Another alternate solution in [10, 9, 16] proposed OS scheduler modifications to map tasks that share data to the same core and thereby avoid simultaneous caching of shared data across the cores. Though these solutions are suitable for real-time systems as they guarantee timing requirements for HRT tasks, they do not offer significant performance benefits. Recently, a hardware-based cache coherence protocol to predictably manage accesses to shared data in multi-cores called PMSI [21] was proposed. PMSI improved the average-case performance compared to other predictable approaches by allowing simultaneous cached shared data accesses. Thus cache coherence is a good solution to provide performance benefits for real-time systems. However, PMSI does not differentiate between different criticality tasks. It provides strict timing guarantees for all tasks (HRT, FRT or SRT). As a result, a FRT or SRT task would cost HRT tasks an opportunity for tighter worst-case latency bounds by providing unnecessary strict timing guarantees for FRT or SRT tasks. Furthermore, PMSI does not offer any mechanisms to encourage FRT and SRT tasks to improve

their average-case performance. Hence, it is beneficial to provide support that allows simultaneous caching of shared data in a multi-core while being criticality-aware, such that HRT tasks have worst-case bounds, and FRT and SRT tasks are provided with performance benefits.

In this thesis, I propose HourGlass, a predictable cache coherence protocol that is criticality-aware. Specifically, HourGlass differentiates between HRT and FRT or SRT tasks and ensures that different requirements of these tasks are met. For HRT tasks, HourGlass ensures worst-case latency bounds. For FRT tasks, HourGlass provides timing guarantees but a looser bound compared to HRT tasks and also support to improve the average-case performance. For SRT tasks, HourGlass provides no guarantees but has support to improve the average-case performance. HourGlass is equipped with a timer-based mechanism to allow for FRT and SRT tasks to improve their performance. The timers allow for a trade-off between potentially improving performance of FRT or SRT tasks while loosening the WCL bounds of HRT tasks. This is acceptable as long as the HRT tasks continue to meet their temporal requirements. Further, HourGlass also has support that allows predictable sharing of data across HRT and FRT or SRT tasks. Most work do not focus on sharing data across different criticality tasks, however, this is conservative. As long as HRT tasks are guaranteed to meet the timing requirements, it is beneficial to support such accesses.

This thesis is organized into several sections as follows: In Chapter 2, I discuss the related work in mixed time-criticality systems and timer-mechanisms in cache coherence protocols. In Chapter 6, I discuss the motivation for HourGlass compared to the existing predictable cache coherence protocol PMSI. I also discuss different versions of HourGlass obtained by changing the criticality-aware bus arbitration. Chapter 5 gives an account of the different arbitration schemes considered in this work. Chapter 7 gives a detailed description of the hardware modifications and cache coherence protocol modifications required for HourGlass. I also provide a timing analysis for all versions of HourGlass, and describe the various latency components that contribute to the WCL of requests from HRT and FRT tasks in Chapter 8. I prototype HourGlass in gem5 [7], a cycle accurate micro-architectural simulator, and evaluate with multi-threaded benchmarks from SPLASH-2 [31] and synthetic benchmarks that exhibit maximum data sharing (Chapter 9). It is observed that the WCL of all memory requests from HRT tasks are within their analytical bounds, and that timers do encourage a trade-off between performance improvement from FRT or SRT tasks and WCL for HRT tasks.

Chapter 2

Related Work

2.1 Real-time systems

Real-time systems ensure that the application does not only function correctly, but also meets the timing requirements. Tasks in the application should complete before a fixed worst-case execution deadline [21]. With increasing demand for real-time systems in several domains with importance to performance and cost-efficiency, there is a shift to multi-core systems. Multi-core real-time platforms contain interconnects, caches, and main memory that are shared by all the cores in the platform. Typical real-time models assume task isolation where there is no sharing of data across the tasks. This requires partioning of the memory based on the number of tasks. Authors in [21] identified that task isolation suffers from limitations such as memory underutilization and does not scale with increasing number of cores. Thus there is need for sharing data across the tasks for performance and cost efficiency. This further complicates the timing analysis for the accesses by the cores due to interference when multiple cores access the shared memory.

Prior work on sharing data across the cores focussed on bypassing the cache hierarchy to access shared data [18, 24]. Alternate solutions proposed in [10, 9, 16] make operating system (OS) changes to include data-sharing aware scheduler policies that identify tasks that share data. The tasks that share data are mapped to the same core and thereby avoid simultaneous access to shared data by multiple cores. Authors in [26] modified the application such that accesses to shared data are protected using lock mechanisms. As a result, shared data is accessed by only one core at any time instance. This approach, in the worst-case will lead to sequential execution of tasks, thereby overriding the effect of parallelism.

Hassan et al. [21] proposed a novel hardware cache coherence protocol Predictable Modified Shared Invalid Cache Coherence Protocol (PMSI) to manage shared data accesses. Compared to prior approaches that either bypass the cache or make application or OS modifications, PMSI allowed simultaneous access to shared data by multiple tasks, resulting in improved average-case execution time. It is a modified version of the standard MSI protocol that allows for sharing of data in a predictable manner. However, this thesis proposes a cache coherence protocol for a different platform of embedded systems - mixed time-criticality systems, that has a combination of hard real-time (HRT) and firm real-time (FRT) or soft real-time (SRT) tasks. PMSI does not differentiate between HRT and FRT or SRT tasks and assumes all tasks to be HRT, thereby providing timing guarantees for all tasks, even though SRT tasks do not require guarantees. Hence, I present HourGlass, that is criticality-aware with support for two time-criticality levels, which offers tighter worst-case latencies for HRT tasks.

2.2 Mixed time-criticality systems

In recent years, work on real-time systems has shifted on allowing tasks of different levels of criticality to share a common hardware platform [6, 8]. Tasks are of different criticality levels based on the levels of safety assurance required by the system (DO 178C and ISO 26262) [8]. If we consider only the timing requirements of the tasks, the system includes a combination of tasks of different time-criticality levels: hard real-time (HRT), firm real-time (FRT) and soft real-time (SRT). For example, consider the automotive and avionics industries. The Anti-lock Braking Systems (ABS) of the automobile (hard real-time task) is strictly required to meet the deadline, otherwise leading to catastrophic behaviour. The automotive navigation system (firm real-time task), allows for infrequent misses to meet the deadline for better average-case performance; however the result is not valid. The entertainment system (soft real-time task), on the other hand, is not required to meet strict timing requirements, but only better average-case performance [5]. Such a system, that has a combination of different time-criticality levels deployed on the same platform, is termed as mixed time-criticality system [15].

Multi-core platform in mixed time-criticality systems form the topic of interest in recent embedded and real-time systems to benefit performance and efficiency. With tasks of different levels of criticality, co-existing on the same platform, allowing data sharing across different levels is an important concern. Most works focussed on disallowing data sharing across different criticality levels in order to maintain isolation of tasks of different criticality levels [8]. However, isolating tasks based on criticality on a multi-core platform is not

trivial, as it requires partitioning mechanisms at the cache and memory level [14]. Also, disallowing data sharing across criticality levels is an unnecessary restriction, that degrades the performance [34]. Alternatively, there are solutions that allow data sharing across criticality levels [14, 10, 34]. As long as the HRT tasks meet the required deadline, there is prospect in supporting data sharing across criticality levels.

In [10], Chisholm et al. studied the tradeoffs caused by data sharing across criticality levels through shared memory and propose techniques to reduce them. They proposed methods to bypass LLC (Last Level Cache) for data accessed by tasks of different criticality levels and accessed data from DRAM, eliminating unpredictable LLC interference across the tasks. They also proposed techniques to assign the tasks that share data to the same core, thereby avoiding concurrent access to the shared data, as a core executes only one task at a time. Giannopoulou et al. in [14] proposed time-triggered scheduling strategy that allows tasks of same criticality to access the shared resources (caches, memory bus) at a time to prevent tasks of different criticality levels from affecting the response time of HRT tasks. Zhao et al. proposed *highest-locker criticality, priority-ceiling protocol* (HLC-PCP) that used semaphores to access shared data. If a task, say SRT, wants to access the data that is shared by a HRT task, then the SRT task acquires the priority of the HRT task till it accesses the data, and then its priority is restored. They also showed that the blocking of shared resources by different criticality levels is bounded and hence there is predictable data sharing across criticality levels. These prior work focussed on adding modifications to the scheduling mechanisms [34, 14] or to the OS [10]. While these mechanisms work well for systems that do not use caches for shared data, for better performance there is a need for a mechanism that makes use of caches for shared data. In this work, I propose a hardware cache coherence protocol along with a predictable arbitration scheme that handles data sharing in a predictable manner in a mixed time-criticality system. I assume a system model similar to [13], where tasks of same criticality are mapped to the same core. Hence the core inherits the criticality of the tasks that are mapped to it.

2.3 Bus arbitration

Bus arbiters for mixed time-criticality systems are different from those for real-time systems, as they must support tasks of different criticality levels in a predictable way. There are several research efforts that provide explicit support for different criticalities in bus arbiters [25, 19, 12, 11]. Paolieri et al. [25] presented a dual-criticality arbitration where HRT tasks (HRT) used round-robin arbitration, and whenever there were no requests from HRT, SRT tasks are serviced yielding a worst-case bound for HRT. Hassan et al. [19] pre-

sented CArb, a statically scheduled two-level bus arbitration scheme that used harmonic weighted round-robin (HWRR) arbitration. The first level performed HWRR amongst criticality classes, and the second among tasks within a criticality class. CArb is not only criticality-aware, but also requirement-aware as it allocates services to tasks based on their requirements. Cilku et al. [12, 11] proposed a two-layer arbiter that distinguished between memory requests from HRT and FRT tasks. Their approach used TDM arbitration such that HRT tasks were pre-assigned slots in the TDM schedule after which a fixed number of slots were reserved to service requests from FRT tasks. Within the slots reserved for FRT tasks, round-robin was performed. Gomony et al. [15] proposed work-conserving arbitration scheme by allocating slack slots to FRT tasks to improve the average-case performance. HourGlass takes inspiration from Paolieri et al. [25], Cilku et al. [12, 11] and Gomony et al. [15]. I consider different combination of arbitration schemes that are criticality-aware and provide varying worst-case latency bounds for the cores, that is explained in Chapter 5.

2.4 Time-based coherence protocol

Several works on timestamp-based cache coherence protocols were proposed such as [27, 28, 32] as alternative techniques to the standard directory-based protocols to reduce network traffic and address the scalability issue of directory protocols. Shim et al. presented Library Cache Coherence (LCC) in [27] where the cache lines are held for a certain time duration, which is maintained by a global timer. In the presence of multiple sharers, a write to a cache line is stalled until the cache lines held by the sharers invalidate after their timer duration. This timestamp-based configuration reduced the network traffic for directory protocols. Singh et al. in [28] proposed time-based cache coherence called Temporal Coherence (TC) to minimize the network traffic overheads for Graphics Processing Unit (GPU) architecture. While LCC focussed on a time-based coherence for a system that implemented sequential consistency, TC focussed on Release Consistency model. TC used timestamp-based fence instructions to avoid the stalling of writes. Yu et al. proposed a novel cache coherence protocol Tardis [32], that is simpler and more scalable compared to the standard directory protocol. Tardis is similar to LCC, but it does not require a globally synchronized clock. It maintains sequential consistency by storing the logical time of the read and write accesses. These prior work on time-based coherence used timers to reduce network traffic for general-purpose multi-core systems and GPU architecture. However, in HourGlass, we use the timer concept to provide differential services to the cores based on their criticality, focusing on providing improved average-case performance for FRT or SRT tasks. A cache line is held for a pre-defined time duration by a core in its private cache,

even if it is requested by other cores, so that repeated reads or writes to the cache line from the core are cache hits, thus improving the performance of FRT and SRT tasks.

Chapter 3

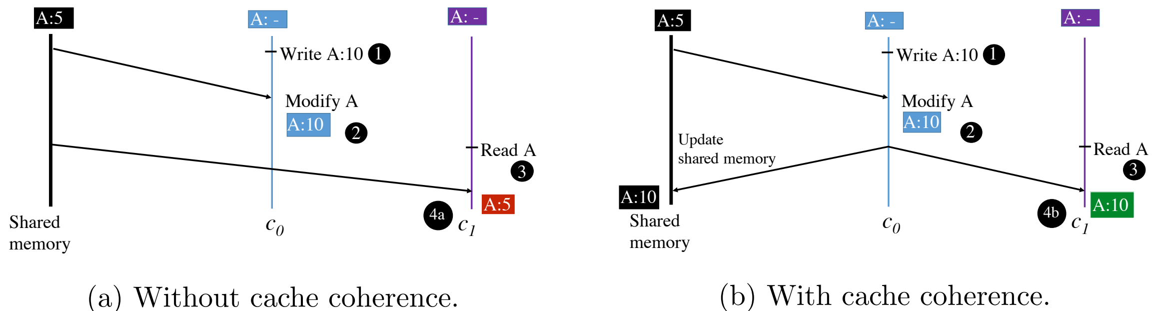
Background

This chapter gives a brief introduction of cache coherence in a multi-core system and the conventional cache coherence protocol. This also provides a background on the mixed time-criticality multi-core system and the bus arbitration schemes used in this work.

3.1 Cache coherence

Multi-core systems share data between cores by accessing addresses within a shared address space. Modern multi-core platforms implement private cache hierarchies that exploit spatial and temporal locality to improve the application's performance. Hence, shared or private data may reside in the private cache hierarchy of multiple cores, allowing multiple copies of the data to co-exist across the cores simultaneously. For correct execution of parallel programs with shared data, it is essential that any core performing a read operation on the shared data obtains the most recent write to the shared data. Since multiple copies of the shared data are privately cached across multiple cores, there must be a mechanism to ensure that reads to the shared data receive the most up-to-date data. This is known as keeping data values *coherent across multiple cores*. A mechanism known as *cache coherence* is a solution that keeps shared data values coherent across multiple cores [29].

Figure 3.1a presents an example illustrating the need for a cache coherence protocol. This example shows two cores c_0 and c_1 accessing shared data `A`. Initially, the shared data variable `A` has a value of `5` in the shared memory and it is not cached in any of the private caches. ➊ Core c_0 performs a write operation on `A` with a value of `10`. This is a miss in c_0's private cache requiring the shared memory to respond with `A` ➋, which is later modified



(a) Without cache coherence.

(b) With cache coherence.

Figure 3.1: Cache Coherence Protocol.

by c_0 with the value of `10`. Next, ❸ core c_1 performs a read operation on cache line `A`, which is a miss in its private cache. Without cache coherence, either the shared memory or c_0 may respond to c_1's request. In order to ensure that c_1 reads the most up-to-date value of the shared data, the correct response should be from c_0's private cache. Figure 3.1a presents a situation without cache coherence where the shared memory responds to the read by c_1 4a. This provides c_1 with a stale value `5` of `A`. This results in the presence of two different copies of the same cache line in the system at the same time. This clearly indicates a need for a set of rules that ensures the value written by c_0 is read by c_1. 4b Figure 3.1b continues the example assuming the existence of a cache coherence protocol. With the presence of a cache coherence protocol, c_0, on seeing a read request from another core, provides c_1 with the latest value `10`. c_0 also writes back the value to the shared memory to maintain a coherent view of `A` across the system.

There are software and hardware techniques to implement cache coherence. Software approaches for cache coherence require additional software instructions to be added to manage the various copies of shared data explicitly by invalidating the data in the cache [2]. These additional instructions impact the performance [23]. It requires the programmer or the compiler to analyze the shared data and invalidate them in a timely manner to ensure the correct functioning of the application [2, 30]. On the other hand, a hardware implementation of cache coherence does not require any application or compiler modifications and is independent of the application. A hardware cache controller implements a protocol that enforces strict rules governing the coherent view of multiple cached copies of data across multiple cores. Hardware cache coherence works at the granularity of cache lines, which is the unit of data transfer in the memory hierarchy. Based on the implementation, hardware cache coherence can be realized either as a *snoopy-bus-based protocol* or a *directory-based protocol* [29]. In a snoopy bus-based protocol, each core broadcasts coherence messages to the shared bus on any read/write request. All the cores snoop the bus to observe the activ-

ity of all the cores. Directory-based cache coherence protocol uses a centralized directory that acts as an interconnect across multiple cores to maintain coherence. The cores and the shared memory communicate only via the directory. This work focuses on hardware cache coherence that use a snoopy shared bus implementation, which is appropriate for current multi-core platforms with eight cores or less (Freescale P4080, [3]) used in real-time multi-core systems.

3.1.1 Coherence protocol

A cache coherence protocol is an implementation of a set of rules that ensures data coherence. This set of rules identifies states that denote the read and write permissions of cache lines, and transitions between these states that occur due to activities of other cores in the system on the same cache line. The **M**odified-**S**hared-**I**nvalid (MSI) cache coherence protocol is a fundamental cache coherence protocol that several modern cache coherence protocols are based upon such as the MESIF, and MOESI protocols [29]. MSI consists of three *stable states*. The semantics for each of these states are as follows: 1) *Invalid* (I) indicates that the cache line does not have valid data. 2) *Modified* (M) represents that the core has modified the cache line data; hence, it has the most up-to-date data. Only one core can have a cache line in the modified state. 3) *Shared* (S) identifies that the cache line was read, but not modified. Multiple cores may have the same cache line in the shared state. This allows read hits in their respective private caches.

Cache coherence protocol also consists of *transient states*, which are intermediate states between stable states. These states represent whether the core is waiting for a data response, or waiting for the memory requests to be ordered on the shared bus. For instance, the transient state IM^{AD} is an intermediate state between the invalid (I) state and modified (M) state, when a write request is issued on the bus by a core with a cache line in I state. Here A denotes that the core is waiting for its coherence message to be broadcasted and D denotes that it is waiting for data response. A cache line changes states based on the activities of the cores. Once the write request is observed on the bus, it moves to another intermediate state IM^D denoting that the core awaits only a data response. Transitions between states occur by exchanging coherence messages between the cores and shared memory.

When discussing coherence activities, I refer to the *private core* to identify the core whose cache controller is under consideration, and *remote core*s as all other cores. Since every private cache implements the same set of rules in its cache controller, I distinguish requests made by the private core and those by the remote cores. To accomplish this, a

core views coherence messages on the bus as either `Own` or `Other` coherence messages. `Own` denotes that the cache controller observes a coherence message generated by its private core, and `Other` as a coherence message generated by remote cores.

MSI Cache Coherence Protocol

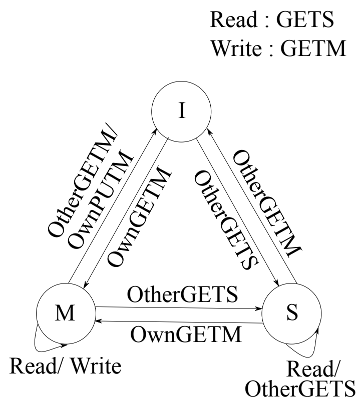


Figure 3.2: MSI Cache Coherence Protocol.

Figure 3.2 shows the state-transition diagram for the standard Modified-Shared-Invalid (MSI) cache coherence protocol. Recall that cores observe the memory activities of other cores in a snoopy bus based cache coherence implementation. Hence, the coherence messages are differentiated as `Other` and `Own` to distinguish messages generated by the private core versus the remote cores. The labels on the transitions denote requests generated either by the private core (read/write) or by remote cores resulting in coherence messages across the snooping bus. For example, a *GetM(`A`)* coherence message generated on a store request denotes that the private core wishes to perform a write on data `A`. Similarly, a *GetS(`A`)* coherence message is generated on a load request, and denotes a read by the private core on data `A`. A *PutM(A)* coherence message is generated during write-backs of dirty cache lines.

Consider the previous example scenario along with the information on different states as shown in Figure 3.3. c_0 has the cache line `A` in a stable *modified* (M) state once it receives the data from the shared memory ❷. Next, ❸ c_1, on observing a miss in its

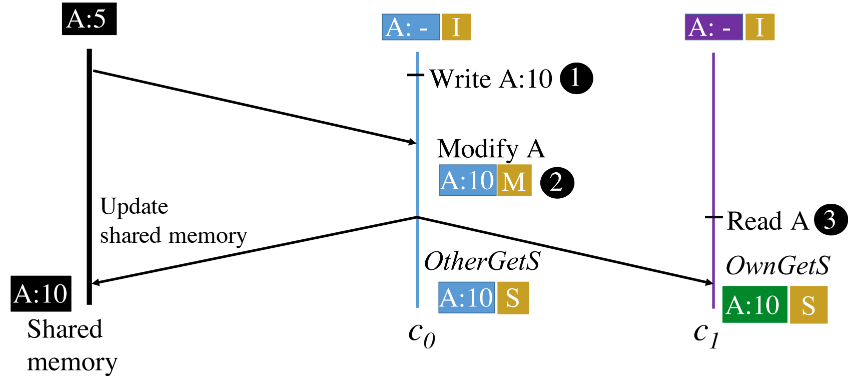


Figure 3.3: Cache Coherence Protocol with states.

private cache for the read operation of `A`, broadcasts the message *GetS(`A`)* on the bus. As a result, c_0 observes this coherence message as *OtherGetS(`A`)* and so responds with the up-to-date value of `A` to c_1 and moves to *shared* (S) state. c_1, on the other hand, observes the coherence message as *OwnGetS(`A`)* and moves to a transient state IS^D. Once it receives the correct data from c_1, c_0 completes the read operation and moves to *shared* state.

3.2 Mixed time-criticality multi-core systems

Recent work on multi-core real-time systems propose tasks of different criticality levels to share the same platform [6]. The tasks are either hard real-time (HRT), firm real-time (FRT) or soft real-time (SRT). These tasks have different requirements, based on their criticality, that must be guaranteed. For example, the deadline by which the task should finish may vary for different criticality levels. Hard real-time (HRT) tasks have strict timing requirements and it is necessary that the tasks complete before the pre-determined deadline. Firm real-time (FRT) tasks, on the other hand, are not required to always meet their deadlines. Instead they can tolerate occasional missed deadlines at the expense of degraded performance. These firm real-time tasks benefit from improved average-case performance. SRT tasks do not have any strict guaranteed requirements, but instead, may require best-effort service such as improved average-case performance. Such a system that has a combination of tasks, that have different timing requirements, are termed as mixed time-criticality systems. I focus on a mixed time-criticality system that consists of a combination of two time-criticality levels: HRT and FRT tasks or HRT and SRT tasks sharing the same platform. The cores in the multi-core system are differentiated as HRT

and FRT or SRT cores, where HRT tasks are mapped to HRT cores and FRT or SRT tasks are mapped to FRT or SRT cores. Figure 3.4 shows a four-core system with two HRT cores and two FRT cores. They are connected via a snoopy shared bus to the memory. Here shared memory represents the DRAM main memory.

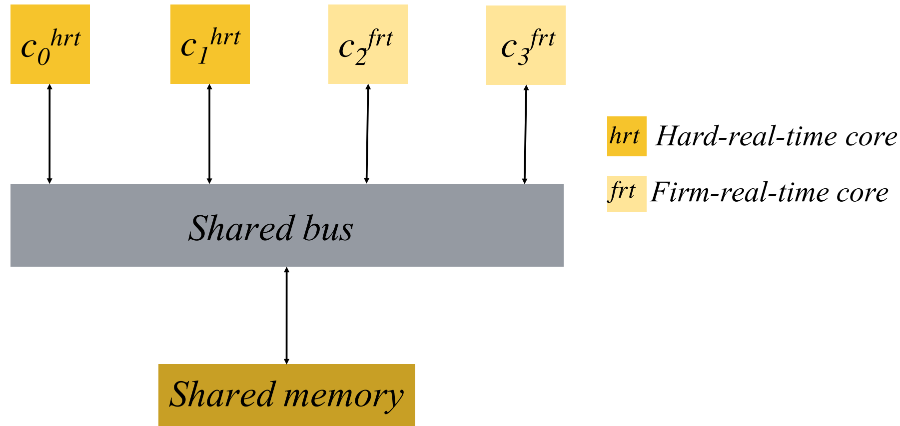


Figure 3.4: Mixed time-criticality multi-core system.

3.2.1 Arbitration schemes

There needs to be a predictable arbitration at the shared bus to decide which task in a real-time application gains access to the hardware resources (for example, the shared memory) via the bus. Time-division multiplexing (TDM) is one of the predictable arbitration schemes that provides timing isolation across the tasks [12] by assigning a dedicated time slot for each core. But with different levels of criticality, different arbitration schemes can be proposed by assigning varying number of guaranteed slots to the tasks based on their criticality. Chapter 2 discusses some prior work on bus-arbiter schemes in [25, 19, 12, 11] to support different criticality tasks to access the shared bus. Another way for a bus arbitration scheme would be to assign all HRT cores with a guaranteed slot, as they are always required to meet their timing requirements. On the other hand, since FRT cores tolerate some misses to timing deadlines, they can be assigned fewer slots and serviced in a round-robin manner. The TDM arbitration can be work-conserving or non-work-conserving. Work-conserving TDM allows a core to use the slot of another core's dedicated slot if the other core has no pending request (slack slot). Non-work-conserving, on the other hand, leaves the slack slots idle. Work-conserving TDM is useful to improve performance as slack slots are used to service other pending requests. HRT cores are required only to

meet the timing requirements, whereas the FRT or SRT cores require better average-case performance. Hence, the slack slots can be used to service requests from FRT or SRT cores to improve their performance [15].

Chapter 4

System Model

This work assumes a mixed-time-criticality multi-core real-time system consisting of a combination of tasks that have two different timing requirements. It assumes the task set to be $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_m\}$ with m number of tasks, where τ_i is a task that has pre-determined requirements to be met depending on its time-criticality and i indicates the task identifier. Based on their timing requirements, the tasks can be hard real-time, firm real-time or soft real-time tasks. Hard real-time (HRT) tasks mandate strict timing requirements and require that they always complete their execution before the deadline. Firm real-time (FRT) tasks, on the other hand, are not required to always meet their deadlines. Instead they can tolerate occasional missed deadlines at the expense of degraded performance. This allows firm real-time tasks to leverage techniques that improve its performance. Soft real-time (SRT) tasks are those that do no require any strict timing guarantees, but only benefit from improved average-case performance. I assume that the system consists of a combination of 2 levels of time-criticality - cl_1 and cl_2, where cl_1 are HRT tasks and cl_2 are either FRT or SRT tasks. It can either have FRT tasks or SRT tasks along with HRT tasks based on the requirements of the second level for that application.

The multi-core platform has N in-order cores $\{c_0, c_1, \ldots, c_{N-1}\}$. Each core has its own private instruction cache (L1-I$) and data cache (L1-D$). The shared memory is assumed to be the main memory. The cores and the shared memory are interconnected with a snooping bus, which exchanges cache coherence messages between them. Additionally, there is a common cache data bus connecting the cores to support cache-to-cache transfers. It is assumed that once a data transfer over the bus starts, it cannot be preempted.

This work assumes that a single task is mapped onto a single core for the duration of the application's execution. Each core is associated with a logic that programs whether

the core is a HRT, FRT or SRT core, based on the criticality of the task mapped to it. Each core in the multi-core system is characterized by the criticality of its task (cl) and the core identifier, c_i^{cl} where $cl \in \{\mathsf{hrt}, \mathsf{frt}, \mathsf{srt}\}$ and i indicates the core identifier. Memory request from HRT cores must have WCL guarantees per request whereas memory requests from FRT or SRT cores have no strict timing guarantees. However, FRT and SRT cores benefit from improved average-case performance. I denote the number of HRT cores as N_{hrt}, FRT cores as N_{frt} and SRT cores as N_{srt}. Since the system assumes only two levels of criticality, it is seen that $N = N_{cl_1} + N_{cl_2}$, where $cl_1 \in \mathsf{hrt}$ and $cl_2 \in \{\mathsf{frt}, \mathsf{srt}\}$. The number of HRT and FRT or SRT cores are set based on the number of tasks of different criticality required for the application. Each core is then programmed to be HRT, FRT or SRT based on the N_{hrt}, N_{frt} or N_{srt}. HRT tasks are mapped to cores that are programmed as HRT, whereas FRT or SRT tasks are mapped to FRT or SRT cores. Hence, the core acquires the criticality of the task that is mapped to it and so tasks and cores are used interchangeably in this thesis. The tasks distributed across the multiple cores with different criticality levels can share data.

Figure 4.1 illustrates a system model assumed for this work. It consists of two HRT tasks and two FRT tasks, mapped on a four-core system. Hence two cores are programmed as HRT and two cores as FRT. Each core has a private L1 cache and the cores and shared memory are connected via the snoopy bus and the common data bus. The shared memory is the DRAM main memory.

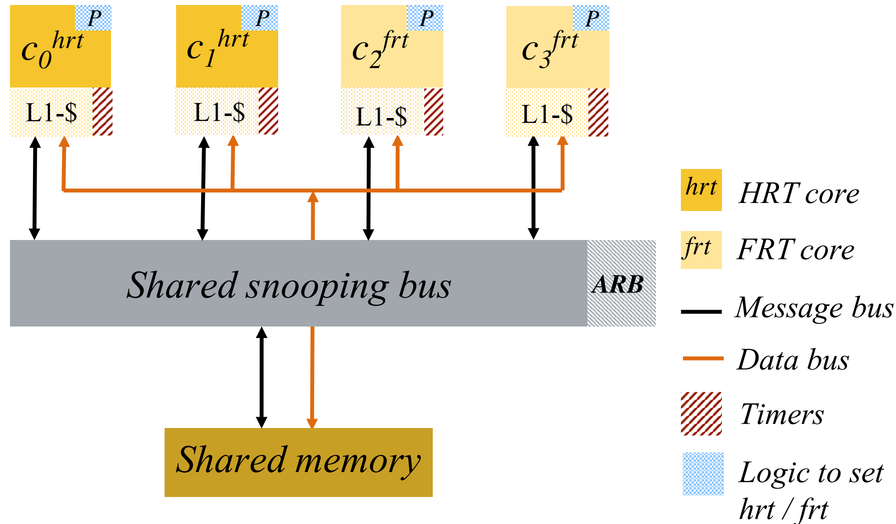


Figure 4.1: Mixed time-criticality multi-core system.

The shared snooping bus uses a predictable arbitration scheme to arbitrate accesses to the shared memory from different cores. I assume Time Division Multiplexing (TDM) for predictable arbitration of memory accesses, although HourGlass can be extended to support any other predictable arbitration scheme. Some TDM slots are pre-assigned for certain cores, and these are termed as *dedicated* slots. In the *dedicated* slots, only the core that is assigned that slot can make a request. If the dedicated core has no pending request, then that slot is idle and is called a *slack* slot.

Chapter 5 describes the different arbitration schemes considered in this work. ALL-DD allocates *dedicated* slots for all the cores, irrespective of whether they are HRT, FRT or SRT. H-DD-NWC, in order to get tighter worst-case latency bounds for HRT cores, allocates *dedicated* slots for all the HRT cores, but only fewer *dedicated* slots to FRT cores. Among the few *dedicated* slots, the FRT cores are serviced in round-robin and the *slack* slots remain idle. To improve the performance of FRT cores, H-DD-NWC is modified to allow the *slack* slots to be utilized by the FRT cores to issue requests. This scheme is given by H-DD-WC. A special case of H-DD-WC is where no slots are dedicated to the second criticality level. This arbitration, H-DD-WC-0, can be utilized for SRT cores, that do not require a guaranteed bound. All HRT cores are assigned *dedicated* slots, whereas SRT cores use only the *slack* slots of HRT cores. Based on the arbitration scheme used, different version of HourGlass is chosen. In order to characterize this, I represent HourGlass as a function of the arbitration scheme chosen (arb) - HourGlass(arb). The hardware for arbitration is present at the shared bus, as shown in Figure 4.1. The bus arbitrates memory requests, data responses, and coherence messages.

I assume the TDM slot-width SW to be large enough to complete one data transfer from the shared memory to the private cache of a core, and the transfer of any necessary coherence messages. The transfer of coherence messages and data responses begin at the start of a TDM slot. I also assume that a core can issue only one memory request in its designated TDM slot. Further, a core is allowed to have only one pending request to the bus. The bus arbiter also supports fixed-priority arbitration to handle multiple pending requests from different cores. Requests from HRT cores are prioritized over the requests from FRT and SRT cores. Multiple pending requests from cores of the same criticality are serviced in First-Come First-Serve (FCFS) manner.

Each core contains timers that enable holding onto the cache line for a fixed time duration (cycles) even when it is requested by a remote core as shown in Figure 4.1. Once the timers expire, the private core responds to the coherence messages from the requesting remote cores. Two timers are associated with each cache line for a core. The countdown timers are denoted as $t_n(A, c_n^{cl}, c_m^{cl})$ where t_n identifies the current value of the timers for a cache line A present in c_n where $n, m \in \{0, 1, ..., N-1\}$ and $cl \in \{\mathsf{hrt}, \mathsf{frt}, \mathsf{srt}\}$. c_n denotes

the core that has a valid copy of the cache line, and c_m denotes the requesting core. For example, assume that c_i is a HRT core, and c_j is a SRT core. Then, $t_i(A, c_i^{\mathsf{hrt}}, c_j^{\mathsf{srt}})$ denotes the timer configuration for cache line A when a HRT core c_i^{hrt} has the cache line A, and observes a memory request from a SRT core c_j^{srt} for the same cache line A. Since there are two time-criticality levels - cl_1 (HRT) and cl_2 (frt or srt), a valid cache line in a cl_1 core, $c_i^{cl_1}$, may receive requests for the cache line from a core of either criticality (cl_1 or cl_2) resulting in two timer configurations $t_i(A, c_i^{cl_1}, c_k^{cl_1})$, and $t_i(A, c_i^{cl_1}, c_l^{cl_2})$ where A is the cache line that is requested, c_k is a HRT core and c_l is either a FRT or SRT core. Notice that these two timer configurations identify the requesting core to be either of the two criticality levels. Similarly, a request to a valid cache line in a core of cl_2 criticality $c_j^{cl_2}$ also has two timer configurations $t_j(A, c_j^{cl_2}, c_i^{cl_1})$, and $t_j(A, c_j^{cl_2}, c_l^{cl_2})$ where c_l is another cl_2 core and A is the cache line. The timers are initialized with timeout values on receiving a valid copy of the cache line. I denote $v(\mathsf{hrt}, \mathsf{hrt})$ as the initial timeout value for a cache line present in the private cache of a HRT core, and requested by another HRT core, and $v(\mathsf{hrt}, cl_2)$, where cl_2 is either FRT or SRT, as the initial timeout value for a cache line present in the private cache of a HRT core, and requested by a FRT or SRT core. Similarly $v(cl_2, \mathsf{hrt})$ and $v(cl_2, cl_2)$ represent the initial timeout values for cache lines present in the private cache of a cl_2 core, and requested by another HRT core and cl_2 core respectively, where cl_2 is either FRT or SRT. The timer configurations are set based on the requirements of the application, and they are not changed during application execution. Moreover, the timers are private to the core; hence, their values are not communicated to other cores.

In HourGlass, I assume that lock variables are not shared between the two criticality levels (HRT and FRT or HRT and SRT). Though HourGlass guarantees per request worst-case latency for HRT tasks, it is necessary to guarantee that the critical section of the task is also bounded in order to have a guaranteed bound for the task. There can be a case when a SRT core obtains the lock and does not release it as the requests are not predictable for SRT tasks. Other mixed criticality work use wait-free buffers to handle this. With the provision of timers in HourGlass, HourGlass can be extended to support sharing of locks across criticality levels. A core, on obtaining a lock variable, holds it for a particular time duration. This time duration can be set based on the upper bound on the duration of critical section for all HRT tasks. If SRT cores still hold the lock variables after this time duration, then they are released, thus providing guarantees for HRT tasks. In this thesis, for experimentation, I assume that FRT and SRT cores have an upper bound on the critical section that requires locking the mutex variables. Thus, this guarantees bounds for HRT tasks.

Chapter 5

Bus Arbitration schemes

The cores in a multi-core platform access the shared memory and DRAM via the interconnecting shared bus. Read/write requests or cache coherence messages are broadcasted to other cores and shared memory using the shared bus. A bus arbiter arbitrates requests from all the cores, such that no core is starved from accessing the bus. For this, there needs to be a predictable bus arbitration scheme in order to avoid unbounded interference from different cores accessing the bus. HourGlass uses Time Division Multiplexing (TDM) for predictable arbitration of requests across the cores. TDM allocates slots of fixed time duration to each core in order to access the shared bus. It provides temporal isolation across the cores. I discuss different arbitration mechanisms that can be used in mixed time-criticality systems, that have different timing requirements for the tasks/cores. Memory requests from HRT cores must have worst-case latency (WCL) guarantees per request, whereas memory requests from FRT cores tolerate infrequent violations of timing guarantees. SRT cores, on the other hand, do not require strict timing guarantees. However, FRT and SRT cores benefit from improved average-case performance. In this thesis, for analysis and evaluation, I allocate only one slot per core in a TDM schedule. However, each core can be allocated more than one slot in a TDM schedule. Figure 5.1 illustrates the allocation of TDM slots to the cores for all the arbitration schemes discussed here. The example in the figure considers a system with four cores with two HRT cores and two FRT or SRT cores. As discussed in Chapter 4, the total number of cores in the system is represented as N, number of HRT cores as N_{hrt}, number of FRT or SRT cores as N_{frt} or N_{srt}.

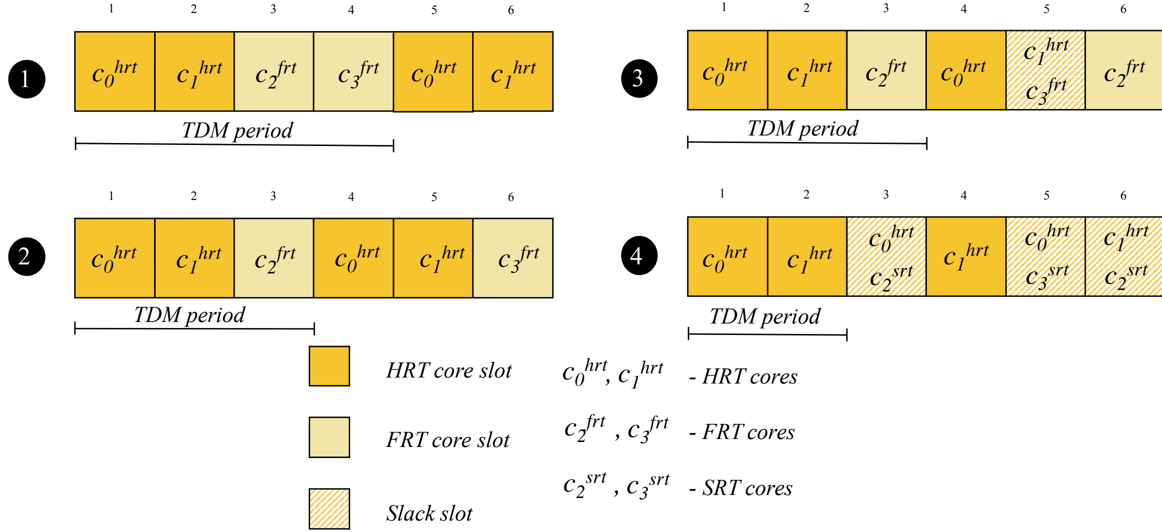


Figure 5.1: Bus arbitration schemes.

5.1 Dedicated slots for all cores (ALL-DD)

ALL-DD allocates a *dedicated* TDM slot for each core, irrespective of its time-criticality. Assuming that the system has two levels of time-criticality with HRT and FRT cores, every HRT and FRT core is guaranteed a *dedicated* slot. Figure 5.1 ❶ illustrates the allocation of slots using ALL-DD. The TDM repeats its schedule after this fixed allocation of slots to cores. The TDM period P is given by,

$$P = N \times SW$$

where N is the number of cores in the system and SW is the fixed slot width.

In Figure 5.1 ❶, each core is allocated a dedicated slot and TDM period P is 4 TDM slots. The TDM slot width SW is large enough to broadcast necessary coherence messages and complete one memory transfer. The worst-case arbitration latency (WCL_{arb}) for any core is given by the time the core takes to gain access to the bus again after it has just missed its slot. The worst-case arbitration latency for HRT and FRT cores are as follows:

$$\begin{aligned} WCL_{arb}^{\text{hrt}} &= N \times SW \\ WCL_{arb}^{\text{frt}} &= N \times SW \end{aligned} \tag{5.1}$$

In the worst-case, the request issued by a core c_i just missed its slot; hence, it has to wait for its next slot, which is after a TDM period. In ALL-DD, HRT and FRT cores have the same worst-case arbitration latency. This arbitration scheme makes no distinction between HRT and FRT cores, and this is a good policy for a real-time system with cores of same criticality level (for example, PMSI). However, for a system with different criticality levels, the arbiter should also be criticality-aware. HRT cores are required to have tight bounds, whereas FRT cores require performance improvements compared to guaranteed bounds. ALL-DD allocates *dedicated* slots for each FRT core, thus treating the FRT core as a HRT core. FRT cores are provided with strict timing guarantees which is not necessary; they benefit from improved performance. To this end, I discuss another arbitration scheme, H-DD-NWC.

5.2 Dedicated slots for HRT cores with Non work conserving arbitration (H-DD-NWC)

H-DD-NWC is criticality-aware and assumes a dual-layer arbiter similar to [12], with a fixed priority arbitration that gives priority to HRT cores. It allocates a *dedicated* slot for each HRT core, and few fixed number of *dedicated* slots to FRT cores. As a result, the WCL for HRT cores is tighter when compared to ALL-DD. Consider the number of slots allocated for all FRT cores be denoted as N_s^{frt}, where $N_s^{\mathsf{frt}} < N_{\mathsf{frt}}$. Among the slots for FRT cores, Round-Robin (RR) arbitration is used, that ensures FRT cores are not starved. RR provides services to FRT cores based on pending FRT requests. If a FRT core has no pending request, then the next FRT core in the round-robin order is given access to the bus. This helps with the average-case performance improvement for FRT cores. In this arbitration, if the dedicated core does not have any pending request to be broadcasted on the bus, then that slot remains idle. Such an arbitration is non work conserving and the idle slot is termed as *slack* slot. The TDM period P for this arbitration is given by

$$P = (N_{\mathsf{hrt}} + N_s^{\mathsf{frt}}) \times SW$$

In Figure 5.1 ❷, 2 slots are allocated to two HRT cores and 1 slot is allocated for FRT cores. The TDM period P is thus 3 TDM slots. Worst-case arbitration latency for the cores is given as follows:

$$\begin{aligned} WCL_{arb}^{\mathsf{hrt}} &= (N_{\mathsf{hrt}} + N_s^{\mathsf{frt}}) \times SW \\ WCL_{arb}^{\mathsf{frt}} &= \left\lceil \frac{N_{\mathsf{frt}}}{N_s^{\mathsf{frt}}} \right\rceil \times P \end{aligned} \tag{5.2}$$

where P is the TDM period.

All the HRT cores are guaranteed a *dedicated* slot, and hence the worst-case arbitration latency for HRT cores is the TDM period. However, only N_s^{frt} slots are allocated for FRT cores per TDM period. The remaining FRT cores are granted access to the bus in the next TDM period and hence WCL_{arb}^{frt} is in terms of P.

5.3 Dedicated slots for HRT cores with Work-conserving arbitration (H-DD-WC)

The TDM arbitration in ALL-DD and H-DD-NWC assume non-work-conserving arbitration, where *slack* slots (no pending request in a *dedicated* slot from its dedicated core) remain idle. From experiments, it is observed that the HRT cores have low slot utilization resulting in more *slack* slots. Work-conserving TDM allows a core to use the slot of another core's dedicated slot if it is a *slack* slot and thus improves performance. However, HRT cores are needed only to meet guaranteed timing requirements, but not improved performance. The FRT and SRT cores, on the other hand, benefit from improved average-case performance. Hence, it makes sense to allow these cores to make use of the *slack* slots to improve their average-case performance.

FRT cores allow for occasional violation of worst-case latency bounds, but it is required to meet the bound for valid outcome. Hence, I assume an arbitration that considers the same policy as H-DD-NWC, along with work-conserving arbitration. All the HRT cores are allocated *dedicated* slots, whereas fixed number of *dedicated* slots are allocated to FRT cores. However, the *slack* slots of HRT cores are utilized to service requests from FRT cores. ❸ in Figure 5.1 illustrates this scheme. In the 5th TDM slot, c_1^{hrt} has no pending request and hence this *slack* slot is used by FRT core c_3^{frt}. By RR, the arbiter grants the $6th$ TDM slot to the next FRT core, c_2^{frt}.

The TDM period P and the worst-case latency bound for HRT and FRT cores are the same as H-DD-NWC given by Equations 5.2. In the worst-case, there are no *slack* slots and hence the FRT cores are issued only in the *dedicated* slots.

SRT cores, on the other hand, do not require any guaranteed timing requirements. SRT cores only require better performance. Hence, I consider a special case of H-DD-WC, where *dedicated* slots are allocated only to HRT cores. I term this arbitration where no slots are dedicated to SRT cores as **H-DD-WC-0**. By not allocating *dedicated* slots for SRT cores, the TDM period is less and thus the worst-case latency bound for HRT cores is tighter.

Dedicated TDM slots are allocated only to HRT cores and the SRT cores are granted access only in *slack* slots of HRT cores.

Now the TDM period P is given by

$$P = N_{\mathsf{hrt}} \times SW$$

❹ in Figure 5.1 shows an example using H-DD-WC-0. Since there are only 2 HRT cores, the TDM period is 2 TDM slots. Worst-case arbitration latency is given by

$$WCL_{arb}^{\mathsf{hrt}} = N_{\mathsf{hrt}} \times SW \tag{5.3}$$

The SRT cores, will not have a guaranteed bound, as they are not allocated a *dedicated* slot. The SRT cores access the bus depending on the availability of *slack* slots.

Chapter 6

Use of Criticality-aware Bus Arbitration and Timers - an Illustration

Cache coherence maintains a coherent view of multiple copies of shared data by propagating changes of one copy to the other copies systematically. However, conventional cache coherence protocols [29] do not distinguish between different time-criticality levels across the cores. Also, conventional cache coherence protocols lead to unpredictable scenarios, which is not suitable for real-time systems that require strict timing guarantees [21]. In [21], the authors propose invariants for predictable cache coherence and propose a protocol PMSI. However, in mixed-time-criticality systems, HRT, FRT and SRT cores have different timing requirements. Recall that HRT cores have strict timing requirements, whereas FRT and SRT cores benefit from improved performance, and allow occasional misses to deadlines. Hence, it is essential to differentiate requests from HRT, FRT or SRT cores in mixed time-criticality systems. It is necessary to reduce the interference suffered by HRT cores originating from other criticality levels to assist in tighter worst-case bounds while encouraging FRT or SRT cores to improve their average-case performance.

PMSI does not differentiate requests from HRT, FRT and SRT cores. Therefore, a FRT or SRT core in PMSI would be treated as a HRT core. This means the coherence state of a shared cache line in a HRT core is affected by the activities of the other criticality level by the mere fact that a TDM slot must be assigned to the FRT or SRT core. Consequently, introducing criticality-awareness in cache coherence can provide tighter worst-case bounds for HRT cores. An alternative is to make modifications to PMSI to support the two different

time-criticality levels considered for that application. Suppose that the bus arbitration policy is changed to distinguish requests from HRT and FRT or SRT cores. H-DD-NWC and H-DD-WC presented in Section 4 are criticality-aware. In contrast to the TDM arbitration used in PMSI, criticality-aware arbitration pre-allocates few slots (in case of FRT tasks) or no slots (in case of SRT tasks) to obtain tighter WCL bounds for HRT cores and allows FRT or SRT cores to utilize the *slack* slots in round-robin fashion to improve their performance. Chapter 5 gives a detailed account of the different arbitration schemes and their benefits over the conventional TDM arbitration.

Assume that PMSI uses one of the criticality-aware arbitration schemes. The FRT cores do not always get a *dedicated* slot every TDM period when H-DD-NWC or H-DD-WC is used. In case of H-DD-WC-0, the SRT cores are never allocated a *dedicated* slot. Hence conventional PMSI will not be predictable with this arbitration, when a HRT core requests for a cache line that is held by a SRT core. This is explained in detail in Chapter 7. So, PMSI requires architectural and coherence protocol changes in order to support this criticality-aware bus arbitration. For the sake of simplicity, let us assume that PMSI also supports cache-to-cache transfers between cores, which it originally does not as presented by Hassan et al. [21]. Figure 6.1 illustrates this implementation of PMSI using H-DD-WC-0 arbitration, that provides for tighter worst-case latency (WCL) bounds for HRT cores by not allocating any TDM slots to SRT cores.

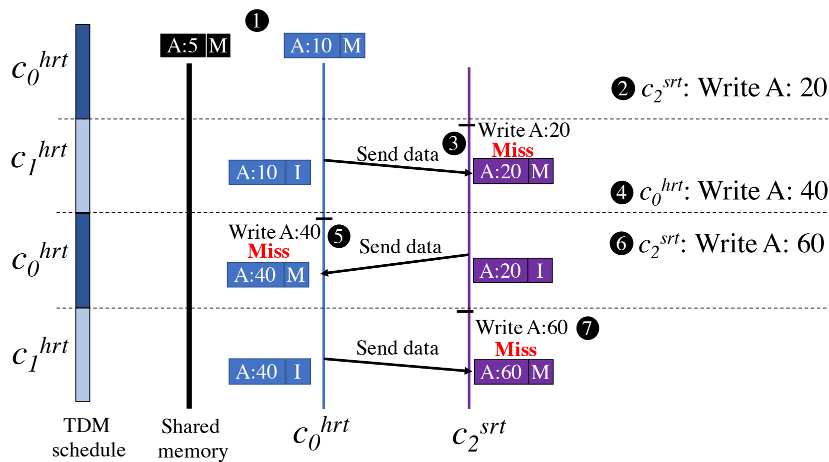


Figure 6.1: PMSI cache coherence protocol with fixed priority arbitration.

Figure 6.1 assumes a four-core multi-core system of HRT cores c_0^{hrt}, c_1^{hrt}, and SRT cores c_2^{srt}, c_3^{srt}. It shows the core activity for HRT core c_0^{hrt} and SRT core c_2^{srt}. Initially, c_0^{hrt} has

data A in modified state ❶. At ❷, c_2^{srt} has a write request. Since, c_1^{hrt} does not have a pending memory request, the arbiter uses the slack slot to issue c_2^{srt}'s write request to A ❸. Since c_0^{hrt} has the most up-to-date copy of A, it sends A to c_2^{srt}, and invalidates its copy ❸. At ❹, c_0^{hrt} has a write request to A. At ❺, c_0^{hrt}'s write request is placed on the shared bus, and c_2^{srt} sends the up-to-date data to c_0^{hrt}, and invalidates its copy. At ❻, c_2^{srt} has a write request to A. Since, c_1^{hrt} does not have a pending request, the bus arbitration grants this slack slot to c_2^{srt}'s pending write request, and completes its write request at ❼. Note that this example performs coherence correctly, but it requires only two TDM slots to be assigned to the HRT cores, as only the HRT cores require timing guarantees. With conventional PMSI, all four cores would need to be assigned at least one slot. Although this is a simple example, it illustrates that one can obtain tighter WCL bounds on HRT cores by using a criticality-aware bus arbitration.

Note that H-DD-WC-0 arbitration only ensures WCL bounds for HRT cores, and no guarantees on WCL or performance for SRT cores. The requests from SRT cores are satisfied depending on the availability of *slack* slots. It is desirable to provide SRT cores a mechanism to improve its performance while still ensuring WCL bounds for HRT cores. For example, in Figure 6.1, consider that the first write request to A from c_2^{srt} (❷) is succeeded by multiple read and write requests to A from c_2^{srt} (not shown). Currently, requests from HRT cores to the same data results in c_2^{srt}'s copy of A to be invalidated. This in turn results in cache misses for succeeding requests to A for the c_2^{srt}. Further, consider that at ❼, c_1^{hrt} has some pending request and so c_1^{hrt}'s slot cannot be utilized by c_2^{srt}. The multiple read and write requests to A from c_2^{srt} can be satisfied only when c_2^{srt} obtains a *slack* slot. A mechanism that I propose in this work is to allow cores to hold shared data in their private caches for a pre-defined time duration. The time duration still guarantees WCL bounds for HRT cores, but it allows SRT cores to leverage cache hits for repeated data accesses resulting in reduced total memory access latency and thus improving performance.

Figure 6.2 shows the use of timers with the criticality-aware bus arbitration policy: H-DD-WC-0. This assumes that only the SRT cores have timers. The initial states (❶, ❷) are similar to the example in Figure 6.1. At ❸, c_2^{srt}'s write request is satisfied in the slack slot of c_1^{hrt}. The difference from the previous example is that c_2^{srt} holds A for a time duration of 1 TDM period. In other words, at the end of the next slot of c_1^{hrt}, c_2^{srt} self-invalidates its copy of A in its private cache. During this time period, successive write requests from c_2^{srt} to A are cache hits ❺,❻ resulting in reduced memory access latency for SRT cores. During this timer period, the response for the write request from c_0^{hrt} ❹ is deferred until the end of the timer duration or timeout ❼. At the end of the timer duration, c_2^{srt} invalidates its copy of A (❽), and sends it to the first pending HRT core, which is c_0^{hrt}. The use of timers

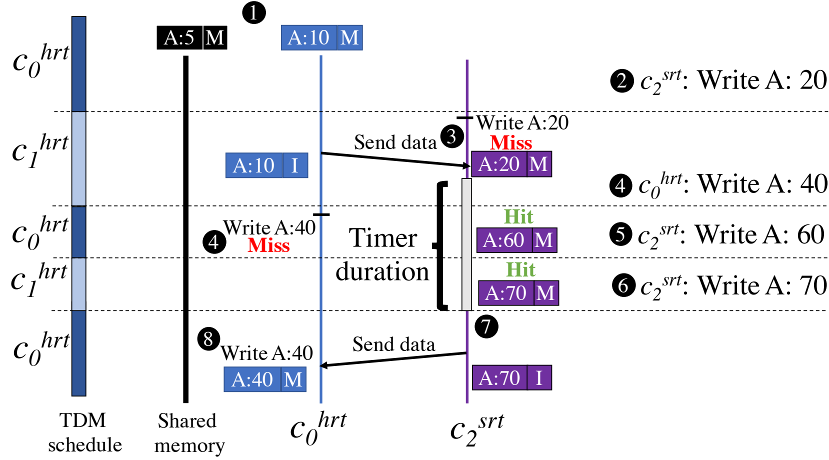


Figure 6.2: Proposed solution with timers for SRT cores.

for SRT cores results in improved average-case performance, while still guaranteeing WCL bound for HRT cores. Note that there is no reordering of memory requests, but there is only a change in the interleaving of requests between two different cores. The memory requests from a single core are still satisfied in order. So the usage of timers is valid under sequential consistency [29]. It is observed that the WCL bound for HRT cores depends on the value of timers set initially for the SRT cores, but it is still a guaranteed bound as the value is constant throughout the progress of the application. The timers, thus, introduce a trade-off between the WCL of memory requests from HRT cores, and performance of SRT cores.

Mixed time-criticality systems also require that there is less interference suffered by HRT cores due to SRT cores. By assuming timers for HRT cores, a HRT core also holds data for some time duration. It gives some buffer time within which HRT and SRT cores can make requests. If there is no HRT core requesting for the same data in that buffer time, then SRT cores can obtain the data. However if there are HRT and SRT cores requesting for the same data, then by fixed priority arbitration, HRT cores are serviced before SRT cores. Figure 6.3 illustrates a scenario where HRT cores also have timers. Initially ❶, c_0^{hrt} holds the cache line A in modified state. At ❷, c_2^{srt} has a write request to A. This write is broadcasted in c_0^{hrt}'s *slack* slot at ❸. Now c_1^{hrt} also requests for write on A, that is broadcasted in the next slot of c_1^{hrt} ❹. If c_0^{hrt} does not hold A for some time duration, then c_2^{srt} would have obtained A at ❸ and holds for some time duration (1 TDM period as in the example in Figure 6.2). Now c_1^{hrt}'s write is interfered by c_2^{srt}'s write and it has to

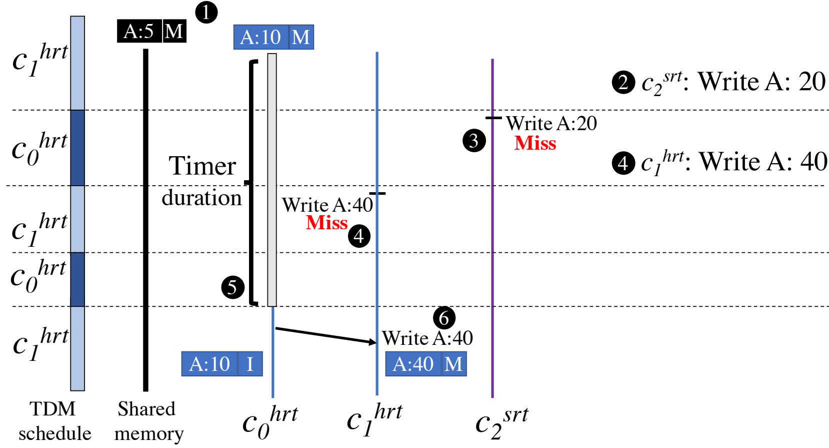


Figure 6.3: Proposed solution with timers for HRT cores.

wait for c_2^{srt}'s timer to expire before it can receive A. Though this will not affect the WCL bound of c_1^{hrt}, it is beneficial to have support to quantify this interference from SRT cores. If c_0^{hrt} holds A for some time duration, say 2 TDM periods, then it expires at the end of c_0^{hrt}'s slot ❺. Now, c_0^{hrt} observes two pending requests, and by fixed-priority arbitration, c_1^{hrt} receives A before c_2^{srt} ❻. Thus, c_1^{hrt} receives A that is not modified by a SRT core, c_2^{srt}. This amount of interference allowed by memory requests of SRT cores on HRT cores can be quantified based on the timer duration. For example, setting the timer duration, that a HRT core holds data, to a high value may prohibit any interference from SRT cores. This is beneficial in applications where it is required to prevent a less secure SRT core from interfering with a HRT core, until it is complete. The presence of timers for HRT cores also lead to increase in number of *slack* slots, as each HRT core should wait for the timers of other HRT cores to timeout. This allows more FRT or SRT cores to access the bus, thereby improving the performance of FRT or SRT cores.

I propose HourGlass that includes all these changes to support criticality-aware fixed priority arbitration and timers. HourGlass is a novel predictable cache coherence protocol compared to PMSI and is useful for mixed time-criticality systems. It provides provisions to improve the average-case performance of SRT cores by increasing the number of cache hits, with an increase in WCL bound for HRT cores.

Chapter 7

HourGlass

HourGlass is a predictable time-based cache coherence protocol for mixed-time-criticality systems. HourGlass is derived from the conventional MSI cache coherence protocol. It is designed such that there is a per request worst-case latency bound for HRT cores as all HRT cores are allocated *dedicated* slots. For FRT cores, few TDM slots are allocated. So there is a per-request worst-case latency bound for FRT cores. It is a looser bound compared to the HRT cores, but it has a guaranteed bound. Since FRT cores utilize the *slack* slots of HRT cores, it also provides improved average-case performance. For SRT cores, no TDM slots are guaranteed and so they do not have a worst-case latency bound, but benefit from improved performance due to the use of *slack* slots. For predictability of any core (observed request latency is always within the derived worst-case request latency bound), HourGlass must satisfy the invariants described in PMSI [21]. However, with different criticality levels co-existing on the same platform, the HRT cores must be prioritized over the FRT or SRT cores. HourGlass requires architectural changes and cache coherence protocol changes to support this criticality-aware cache coherence.

7.1 Architectural Modifications

Figure 7.1 shows the architectural modifications necessary to support HourGlass(arb). There is a need for modifications in the architecture of the shared bus and the cache controller, as explained in Sections 7.1.1 and 7.1.2.

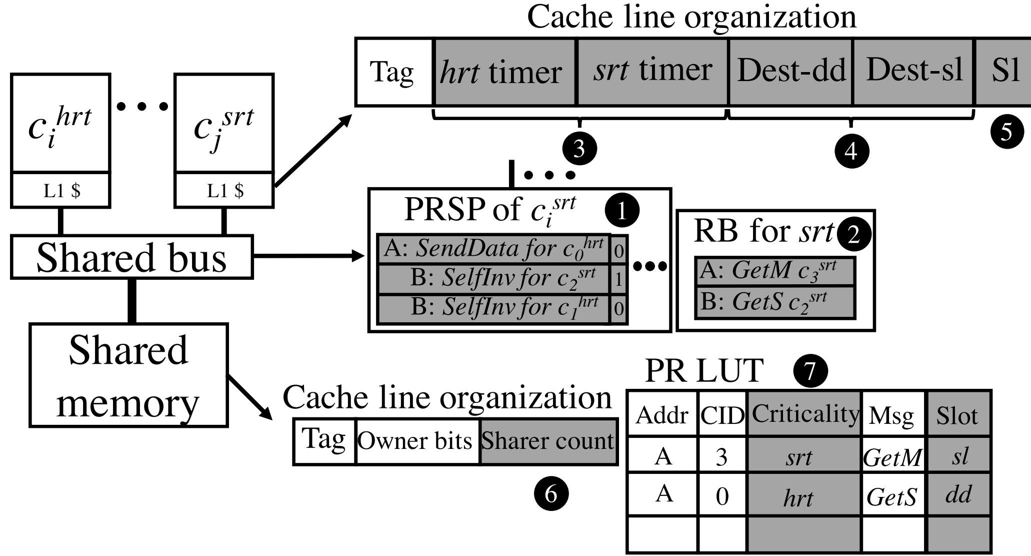


Figure 7.1: Architectural Modifications.

7.1.1 Architectural modifications to shared bus

The arbiter assigns *dedicated* slots to all the HRT cores and few *dedicated* slots to FRT cores or no *dedicated* slots to SRT cores, depending on the arbitration scheme chosen. The *slack* slots of HRT cores are utilized by FRT or SRT cores. I assume that the hardware support to handle this arbitration across the requests from different cores, is already available in the shared bus. For instance, the shared bus can have a pending request queue for each core, from which the arbiter chooses the request from the corresponding core for that TDM slot.

FRT cores can broadcast their requests on the bus in the *dedicated* slots for FRT cores or in *slack* slots of HRT cores. SRT cores broadcast their requests only in the *slack* slots of HRT cores. It is therefore required to identify the FRT or SRT cores utilizing the *slack* slots. This is required because the requests from FRT or SRT cores broadcasted in the *slack* slots lead to unpredictable behaviour for requests from HRT cores. There can be more pending requests (FRT or SRT) than the number of *dedicated* slots in the TDM schedule, leading to unbounded request latency for HRT cores. As a result, it is required that the requests broadcasted in the *slack* slots are cancelled and reissued in the presence of a pending request from HRT core. To this end, the arbiter must append the information on whether it is a *slack* or *dedicated* slot in the coherence message with a bit *isSlackSlot*, before broadcasting it. With this information, HourGlass takes appropriate state transitions.

Recall that for H-DD-WC-0, the arbiter does not allocate *dedicated* slots to the SRT cores. Assume that a HRT core, c_i^{hrt}, requests for a cache line that is present in the private cache of a SRT core, c_j^{srt}. In such a scenario, the predictability of the request from c_i^{hrt} is lost, as there is no guarantee that the c_j^{srt} will definitely get a *slack* slot, where it can send data to c_i^{hrt}. To handle this, HourGlass allows for sending data in the requesting core's slot. To this end, HourGlass requires the addition of two new coherence messages needed for cache-cache transfer (when cache line is in modified state) called *SendData* and self-invalidation (when cache line is in shared state) called *SelfInv*. More details on these coherence messages are provided in Section 7.2. These coherence messages are required to be reordered based on criticality, before they are broadcasted in the requesting core's slot. For example, consider c_0^{hrt} has a cache line A in modified state in its private cache. Next, there are two pending requests to A from c_2^{srt} in *slack* slot and c_1^{hrt} in *dedicated* slot, observed in the same order by c_0^{hrt}. c_0^{hrt} sends *SendData* coherence message initially for c_2^{srt}, and then on observing a higher priority c_1^{hrt}'s request, sends another *SendData* coherence message for c_1^{hrt}. In this case, HourGlass should reorder *SendData* messages such that c_0^{hrt} sends data to c_1^{hrt}, and cancel the pending data response coherence message to c_2^{srt} although c_0^{hrt} observed the request from c_2^{srt} first. To support this, a pending response (PRSP) buffer for each core is added in the shared bus. This buffer holds the coherence messages issued by the corresponding core related to sending data to another core and to shared memory (*SendData*), and self-invalidation messages (*SelfInv*). This is shown as ❶ in Figure 7.1. Each PRSP buffer is of size N. Recall from the example that, in case of multiple pending requests from HRT and SRT cores, the core that has the cache line issues two coherence messages. Hence, whenever a coherence message for a HRT core is added to the PRSP buffer, the arbiter cancels the pending coherence messages for requests from FRT or SRT cores broadcasted in *slack* slots. These cancelled messages are identified by using a valid bit in the PRSP entries as shown in ❶. When a coherence message for a particular data is added to the PRSP buffer, that already has a valid pending coherence message for the same data but for a different core of same criticality level, then the new coherence message is cancelled. This is done to ensure that requests from the same criticality level are serviced in order in first-come first-serve policy.

Adding PRSP buffers requires changes to the criticality-aware arbitration logic. At the start of a TDM slot for a particular core c_k, the arbiter needs to check the PRSP buffers of all the cores if there exists a valid pending coherence message (*SendData* or *SelfInv*) for c_k. If there exists one, then this message is broadcasted in the TDM slot for c_k. HourGlass allows for the existence of multiple sharers of a cache line simultaneously. As a result, for a core c_k (requesting for a write operation), there can be $N-1$ cores holding the cache

line in shared state. Hence, there are now $N-1$ *SelfInv* messages for c_k sent by all the cores in shared state. The arbiter allows for the broadcast of multiple coherence messages for the same core. Thus, the slot width must be large enough to account for the transfer of at least $N-1$ coherence messages and latency due to data transfer from main memory. If c_k does not have any pending request, or any coherence message in its PRSP buffers, then this TDM slot (*slack* slot) is assigned to a FRT or SRT core in round-robin. The arbiter determines which FRT or SRT core to grant access to the bus at the beginning of the slot, and only one core may proceed in that slot. This ensures that the core completes its request within the slot.

Recall that FRT or SRT requests broadcasted in *slack* slots are reissued on observing a HRT request. These reissued requests are again broadcasted in the next slot for that FRT or SRT core. The SRT cores do not require guaranteed bound. However, in order to have a bound for FRT cores, the reissue request from a FRT core is broadcasted in a *dedicated* FRT slot (detailed explanation in Section 7.2.1). To handle this, the shared bus has a Reissue Buffer (RB) shown as ❷ in Figure 7.1, that holds reissue requests from FRT cores. In the *dedicated* slot of a FRT core, the arbiter first broadcasts the requests from RB buffer, and if it is empty, it broadcasts requests from FRT cores in round-robin.

7.1.2 Architectural modifications to cache controllers

HourGlass uses timers to guarantee WCL bounds for HRT cores while simultaneously allowing FRT or SRT cores to benefit from improved performance. These timers are typically available in most current commercial-off-the-shelf (COTS) micro-architectures in the form of time-stamp registers. For example, x86 architectures have the timer-stamp registers [22] that increments based on the core frequency, and most ARM architectures have a system counter that counts at a specific frequency [4]. HourGlass utilizes these commonly found high precision timer support to deliver WCL bounds for HRT cores, and performance improvement for FRT and SRT cores.

In order for cores to hold cache lines for a time duration, the tag bits of cache lines are extended to hold two timer values based on their criticality. For example, each cache line A in HRT core, c_i^{hrt}, will have two countdown timers $t_i(A, c_i^{\mathsf{hrt}}, c_j^{cl_2})$, where cl_2 is either FRT or SRT (FRT or SRT timer) and $t_i(A, c_i^{\mathsf{hrt}}, c_k^{\mathsf{hrt}})$ (HRT timer) where $i, j, k \in \{0, ..., N-1\}$. The initial values of the countdown timers are set based on the latency requirements of HRT tasks and on the application. The counter values are represented as 64 bit values. This architectural modification is shown in Figure 7.1 as ❸. When a core receives a cache line from either the shared memory or from a remote core, the timer values are set with

the timeout configured values, and begin to count down every cycle until they are zero. When TDM arbitration is used, the initial values of the timer can be set in terms of TDM periods. In this case, the timers are aligned to TDM slots and expire at the end of TDM periods. Now, the bit overhead per cache line can be reduced to 12 bits by storing the TDM period at which the cacheline is invalidated.

The cache controller is also augmented with logic to identify the criticality of the remote requests. The cores are aware of the criticality of other cores present in the multi-core platform. This information can be configured at boot time, and stored in a dedicated on-chip read-only memory (ROM). The criticality of the requesting core and information on the issuance of the request in *slack* or *dedicated* slot, is required for the coherence protocol to make correct transitions and prevent unpredictability. This allows a core that holds a valid copy of a cache line to differentiate criticality of pending requests, that is needed for cache-to-cache transfer. The core that holds a valid copy of cache line, marks the requesting core's identifier in a field called *Dest*. This is required for the core to send its data to the requestor after its timer expires. I extend the tag bits of a cache line to include two destination fields to denote the requests broadcasted in *slack* slots (`Dest-sl`) and *dedicated* slots(`Dest-dd`) as shown in Figure 7.1 ❹. The cache controller is included with a logic to identify the core identifier to which the data is transferred after the timers expire. In the presence of multiple pending requests from FRT cores, data is transferred to the first FRT core that made the request, irrespective of whether it was from the *slack* slot or *dedicated* slot. But if there is a pending HRT core, then data is transferred to the core that made the request in a *dedicated* HRT or FRT slot. Any request made by a FRT or SRT core in *slack* slot is reissued in the presence of a HRT request.

Each cache line is also extended with one bit `Sl` to identify if the request for that core is from a *slack* slot or *dedicated* slot shown as ❺ in Figure 7.1. Recall that this information is sent by the arbiter when it services the memory requests. This bit is required by the FRT or SRT cores as they must reissue their request on seeing a HRT core only when it is from a *slack* slot.

In HourGlass, the shared memory keeps track of the number of sharers of a cache line in the system. This is not necessary in conventional cache coherence protocols and PMSI. HourGlass requires this information at the shared memory due to the presence of multiple sharers of a cache line, that timeout at different instances. In order to maintain coherence, only one core can be in modified state at a time. Hence, when a core c_i broadcasts a write request in the bus on a cache line that already has multiple sharers, c_i should wait for all the sharers to invalidate before it can obtain the cache line for write operation. This important property of cache coherence protocols is termed as single writer multiple reader (SMWR) invariant. To ensure this SMWR invariant, HourGlass requires that each entry in

the shared memory be extended with bits to hold the number of sharers of a cache line at that instant (❻ in Figure 7.1). In the worst-case, all cores can have the cache line in shared state and hence it requires $log_2 N$ bits per cache line, where N is the number of cores ($N_{\mathsf{hrt}} + N_{\mathsf{srt}}$).

The shared memory also has a pending request lookup table (PR LUT) similar to PMSI [21], that is used to manage multiple pending requests in a predictable manner. HourGlass, however, requires modifications to the PR LUT such that pending requests to a cache line from HRT cores are serviced before pending requests to the same cache line from FRT or SRT cores broadcasted in *slack* slots. Hence, I extend the PR LUT to include the criticality of the pending requests to a cache line shown as ❼ in Figure 7.1. Pending requests from FRT or SRT cores issued in *slack* slots are cancelled when there is a HRT request to the same cache line.

7.1.3 Hardware overhead

Each cache line in the private caches is extended by 2 timer fields, two destination fields and one `Sl` bit. For a 4-core system, this results in an overhead of $64 + 64 + log_2(4) + log_2(4) + 1 = 133$ bits per cache line. This overhead is for a general solution of HourGlass, where any arbitration scheme can be considered. However, if TDM arbitration is used with the timers values aligned to TDM periods, then 4 bits for each timer in a cache line are enough to store the initial values for timers. Hence the bit overhead is reduced to $4 + 4 + log_2(4) + log_2(4) + 1 = 13$ bits per cache line. Each cache line in the shared memory includes the sharer count, which for a 4-core system results in an overhead of $log_2(4) = 2$ bits per cache line. Each entry in the PR LUT in the shared memory is extended by a bit to denote the criticality of the request and another bit to denote if it was broadcasted in *slack* or *dedicated* slot. Each entry in the PRSP buffer holds the memory address for the request, the coherence message, the core identifier, and a valid bit. For a physical address space of 4GB, this results in roughly 40 bits per entry.

7.2 Cache Coherence Protocol Modifications

HourGlass requires modifications to the MSI cache coherence protocol in addition to the architectural changes described in the previous section. This is because the coherence messages and responses need to be differentiated based on the criticality of the cores. Tables 7.1 and 7.2 describe the different coherence states and transitions between the

State	Core events				Bus events - common to `hrt` and `frt` or `srt` cores								
	Load	**Store**	**Replacement**	**Timeout**	**Own-GetS**	**Own-GetM**	**Own-PutM**	**Own-SelfInv**	**InvAll**	**Own-SendData**	**Data**	**OtherGetS-`hrt`, `frt` or `srt`**	**OtherGetM-`hrt`, `frt` or `srt`**
I	issue GetS /IS^{AD}	issue GetM /IM^{AD}	X	X	X	X	X	X	X	X	X	-	-
IS^{AD}	X	X	X	X	/IS^{D}	X	X	X	X	X	X	-	-
IS^{D}	X	X	X	X	X	X	X	X	X	X	load /S and ST	*	*
$IS^{D}I$	X	X	X	X	X	X	X	X	X	X	load /$S^{T}I$ and ST	*	*
S	hit	/$S^{T}M$ and WT	issue SelfInv /SI^{A}	/S and RT	X	X	X	X	X	X	X	-	update Dest-`sl` or `dd` /$S^{T}I$ and WT
$S^{T}I$	hit	$S^{T}M$ and WT	issue SelfInv /SI^{A}	issue SelfInv /SI^{A}	X	X	X	X	X	X	X	-	update Dest-`sl` or `dd`
SI^{A}	hit	issue SelfInv and GetM /SM^{A}	issue SelfInv	X	X	X	X	/SI	X	X	X	-	issue SelfInv
SI	hit	issue GetM /IM^{AD}	/I	X	X	X	X	X	/I	X	X	-	-
$S^{T}M$	hit	X	X	issue SelfInv and GetM /SM^{A}	X	X	X	X	X	X	X	-	update Dest-`sl` or `dd`
SM^{A}	hit	X	X	X	X	X	X	/IM^{AD}	X	X	X	-	issue SelfInv
IM^{AD}	X	X	X	X	X	/IM^{D}	X	X	X	X	X	-	-
IM^{D}	X	X	X	X	X	X	X	X	X	X	store /M and ST	*	*
$IM^{D}I$	X	X	X	X	X	X	X	X	X	X	store /$M^{T}I$ and ST	*	*
M	hit	hit	issue PutM /MI^{A}	/M and RT	X	X	X	X	X	X	X	/$M^{T}I$ and WT	update Dest-`sl` or `dd` /$M^{T}I$ and WT
$M^{T}I$	hit	hit	issue PutM and SendData /MI^{A}	issue SendData /MI^{A}	X	X	X	X	X	X	X	update Dest-`sl` or `dd`	update Dest-`sl` or `dd`
MI^{A}	hit	hit	issue PutM /MI^{R}	X	X	X	WB/I	X	X	issue Send-Data/I	X	issue SendData	issue SendData

Table 7.1: State transitions at the private cache

states for private caches. Table 7.3 describes the coherence states and state transitions for shared memory controller. Core events denote activity of the core such as loads, stores, and replacements, and bus events denote coherence messages and data responses observed on the bus. In the tables, *msg*/*state* denotes that a core issues the message *msg*, and moves to a coherence state *state*. Cells marked as "×" indicate that a bus or core event cannot happen for a cache line in that state, and cells marked as "-" denote that a cache line in that state does not change state or take any action with a core event or bus event. Shaded rows are new states introduced by HourGlass. In Table 7.1, "WT" denotes wait for timer timeout, "RT" denotes restart timer and "ST" denotes start timer. The modifications to MSI are categorized into two categories: modifications for criticality awareness, and modifications for timer support.

7.2.1 Modifications for Criticality awareness

Recall from section 7.1.2 that the cache controllers have hardware support to differentiate memory requests based on the criticality of the requesting cores and the issuance of the requests in *dedicated* or *slack* slots. In the presence of pending requests from only FRT cores or only from SRT cores, the requests are serviced in First-Come First-Serve policy. However if there is a pending HRT request, to maintain predictability of the HRT request, the first request that was broadcasted in a *dedicated* slot is serviced. FRT or SRT requests broadcasted in *slack* slots are reissued. Since FRT cores are also required to have bounds, the requests from FRT cores that were broadcasted in their own dedicated slot are not reordered. This differentiation in responses to the requests from different cores, is carried out through coherence messages issued to the bus for the remote cores. Since each core is aware of the criticality of remote requestors, the core identifier information in the coherence messages is used to identify the criticality of the incoming request. Each core identifies the coherence messages based on the criticality of the remote core as *Message*-hrt, *Message*-frt and *Message*-srt. For instance, *OtherGetM*-hrt denotes that a core with a valid copy of a cache line observes a write request from a HRT core. Table 7.2 shows the different state transitions at the private cache of the cores, based on the criticality and issuance of requests at *dedicated* or *slack* slots. Further, recall from Section 7.1.1 that the arbiter appends the coherence message with information whether the request is broadcasted in *dedicated* or *slack* slot. Hence the core that made the request, on observing its own message, updates the bit Sl based on the slot (*dedicated* or *slack*) the request was broadcasted. ❶ in Table 7.2 denotes that the Dest-dd is updated only if Dest-dd is empty (this is the first request that is broadcasted in *dedicated* slot). Based on these information, the core with the valid cache line identifies the remote core to which it should transfer data.

The requests broadcasted by FRT or SRT cores in *slack* slots are cancelled in the presence of a pending HRT request. As explained in Section 7.1.2, a FRT or SRT core, that broadcasted its request on a cache line A in *slack* slot, reissues its request on A on observing a HRT remote request on A. In the worst case, the reissue request of a FRT core, say c_j^{frt}, is always broadcasted in a *slack* slot, resulting in unpredictable behaviour for c_j^{frt}. To handle this, any reissue request from a FRT core is broadcasted on the bus in a *dedicated* slot for FRT cores. The FRT core should thus send this information (whether it is a reissue request) along with the coherence message. A bit *isReissue* can be used to identify it. The arbiter then takes appropriate action to broadcast the reissue requests in *dedicated* slots.

In HourGlass, the shared memory is required to identify cache lines that are not cached in the private cache of any core (I state), and when there exists at least one core with a valid read-only copy of the cache line (S state). To identify this differentiation, the shared

State	Sl	Bus events					
		OtherGetS-hrt *dedicated* slot	**OtherGetS-frt** *dedicated* slot	**OtherGetS-frt/srt** *slack* slot	**OtherGetM-hrt** *dedicated* slot	**OtherGetM-frt** *dedicated* slot	**OtherGetM-frt/srt** *slack* slot
IS^D	`dd`	-	-	-	update `Dest-dd`/IS^DI	update `Dest-dd`/IS^DI	update `Dest-sl`/IS^DI
IS^DI	`dd`	-	-	-	update `Dest-dd`❶	update `Dest-dd`❶	-
IM^D	`dd`	update `Dest-dd`/IM^DI	update `Dest-dd`/IM^DI	update `Dest-sl`/IM^DI	update `Dest-dd`/IM^DI	update `Dest-dd`/IM^DI	update `Dest-sl`/IM^DI
IM^DI	`dd`	update `Dest-dd`❶	update `Dest-dd`❶	-	update `Dest-dd`❶	update `Dest-dd`❶	-
IS^D	`sl`	-	-	-	reissue GetS/IS^{AD}	update `Dest-dd`/IS^DI	update `Dest-sl`/IS^DI
IS^DI	`sl`	-	-	-	reissue GetS/IS^{AD}	update `Dest-dd`❶	-
IM^D	`sl`	reissue GetM /IM^{AD}	update `Dest-dd`/IM^DI	update `Dest-sl`/IM^DI	reissue GetM /IM^{AD}	update `Dest-dd`/IM^DI	update `Dest-sl`/IM^DI
IM^DI	`sl`	reissue GetM /IM^{AD}	update `Dest-dd`❶	-	reissue GetM /IM^{AD}	update `Dest-dd`❶	-

Table 7.2: Different state transitions based on core criticality and issuance of requests in *slack* or *dedicated* slots

State	Bus events				
	GetS- `hrt/frt/srt`	**GetM-** `hrt/frt/srt`	**SendData**	**PutM**	**Data**
I	send Data /S	send Data /M	X	X	X
S	send Data /S	add to PR LUT /SM	X	X	X
SM	add to PR LUT	add to PR LUT	X	X	X
M	add to PR LUT	add to PR LUT	update owner/M or wait for data/S_D	wait for data/M_D	X
M_D	X	X	X	X	write to memory /I
S_D	X	X	X	X	write to memory /S

Table 7.3: State transitions at the shared memory

memory has two separate states - I and S state, where I denotes no private copy of cache line in any core and S denotes at least one core has the cache line in shared state. In the conventional MSI cache coherence protocol and PMSI, the shared memory requires no such differentiation and hence fuses the I and S states into one I/S state. This is necessary in HourGlass in order to track multiple sharers of a cache line that can timeout at different time instances. HourGlass also introduces another new state SM that occurs due to a write request on a cache line that already is present in a shared state in at least one core's private cache. Hence, the write request has to wait until all the sharers time out, and SM state denotes this waiting state. Once all the sharers timeout, then it moves from SM to M state by sending cache line to the core that requested for write operation. The shared memory also has information about the criticality of the cores present in the system. This is necessary to identify and reorder requests in the PR LUT and issue data responses to the requesting cores. The different states and state transitions in the shared memory are shown in Table 7.3.

7.2.2 Support for Timers

Timers are required in HourGlass to allow the cores to hold cache lines for a duration of time. HourGlass requires changes to the coherence protocol in order to support the use of timers. New stable and transient states are introduced in HourGlass to accommodate timer support. When a core receives data either from a remote core or from the shared memory, it moves to a stable state (S or M state depending on read/ write request), and populates both the timer fields with the initial timeout values and begins to count down. When a core, that has a cache line in a valid stable state such as S (shared) or M (modified) state, observes a request from a remote HRT, FRT or SRT core on the same cache line, it moves to a stable state S^TI or M^TI. These states indicate that there are pending remote requests for the cache line, but the responses to these requests are deferred until the timers expire. Once the timers expire, the core issues coherence messages to the remote requests based on their criticality and their broadcast in *dedicated* or *slack* slots, with the help of `Dest` fields. If the core has the cache line in shared state (S^TI), then it issues a coherence message *SelfInv* to indicate that the core is ready to invalidate itself. If the core has the cache line in modified state (M^TI), then it issues a coherence message *SendData* to indicate that that core is ready to send data. Once the core issues these coherence messages, they move to S^IA state or M^IA state, indicating that it is waiting for the coherence message to be broadcasted in the requesting core's slot. These coherence messages contain the requesting core identifier, criticality information and information of whether the requestor broadcasted its request in *dedicated* or *slack* slot. The PRSP buffer

holds these coherence messages until it is broadcasted in the requesting core's slot. As a performance optimization, if a core with a valid copy of the cache line does not observe any remote requests, and the timers expire, HourGlass has support to restart the timers.

To maintain the SMWR invariant (described in Section 7.1.2), a write request on a cache line with multiple sharers should wait for the sharers to invalidate. Recall that different sharers invalidate at different time instances. Hence, the write can proceed only after the last sharer invalidates (its timer expires). In this scenario, any core c_i in shared state, whose *SelfInv* message is broadcasted, need not invalidate as it still has the valid cache line. Any future read on c_i can be a hit until the last sharer invalidates. To this end, a state SI is added, that indicates that the *SelfInv* message is already broadcasted, but there are still other sharers of the same cache line. The shared memory issues a coherence message *AllInv* once all the sharers of a cache line invalidate, to notify all the cores to invalidate their copy of the cache line. The shared memory, on observing a *SelfInv* message, decrements the sharer count. When all the timers of all the sharers expire, the sharer count is zero and so the shared memory broadcasts a coherence message *AllInv*, which indicates that all sharers can invalidate and the write can proceed safely.

In HourGlass, for a cache line A, present in a core c_i, to move from S to M state on a write request, it has to wait for its own timer ($t_i(A, c_i^{\mathsf{hrt}}, c_i^{\mathsf{hrt}})$, $t_i(A, c_i^{\mathsf{frt}}, c_i^{\mathsf{frt}})$ or $t_i(A, c_i^{\mathsf{srt}}, c_i^{\mathsf{srt}})$) to expire before proceeding with the write request. This introduces a new stable state $S^T M$, indicating that it is waiting for its own timer to expire. HourGlass currently does not allow for upgrades, which are optimized transitions from S state to M state. When there is a write request on a cache line that is in S state, it can move to M state only after all the sharers are invalidated. However, in HourGlass, the sharers invalidate only after their timers expire and the cores can have their timers expiring at different instances. HourGlass also does not support a cache line in M state to downgrade to S state, on observing *OtherGetS*. However this is not a constraint. HourGlass can support these optimizations with additional transient states and state transitions.

7.2.3 Illustrative Examples

Multiple pending requests from HRT and SRT cores

Figure 7.2 illustrates a scenario with multiple pending requests to a shared cache line A using HourGlass(H-DD-WC-0). This arbitration assumes *dedicated* slots for all HRT cores and no *dedicated* slots for SRT cores. SRT cores utilize the *slack* slots of HRT cores. Initially no core has a valid copy of A. The initial timeout value for all the timer configurations is 2 TDM periods.

TDM slots

CORES	c_0^{hrt}	c_1^{hrt}	c_0^{hrt}	c_1^{hrt}	c_0^{hrt}	c_1^{hrt}	c_0^{hrt}
c_0^{hrt}			❺ Write A: 300 A: $I \rightarrow IM^D$				⓫ Recv A: 100 A: $IM^D \rightarrow M$
c_1^{hrt}							
c_2^{srt}		❸ Write A: 5 A: $I \rightarrow IM^D$	***OtherGetM-hrt(A)*** Reissue Write A:5 ❻ A: $IM^D \rightarrow IM^{AD}$				
c_3^{srt}	Write A: 100 A: $I \rightarrow IM^D$ ❶ / Recv A:50 ❷ A: $IM^D \rightarrow M$	❹ ***OtherGetM-srt(A)*** A: $M \rightarrow M^T I$	❼ ***OtherGetM-hrt(A)*** A: $M^T I$		Read hit to A ❽ ***$t_3(A, c_3^{srt}, c_0^{hrt}) = 0$*** Issue ***SendData for c_0^{hrt}*** ❾ A: $M^T I \rightarrow MI^A$		❿ ***Own_SendData*** Send A:100 to c_0^{hrt} A: $MI^A \rightarrow I$
Shared memory	***GetM-srt(A)*** A: $I \rightarrow M$ Send A:50	***GetM-srt(A)*** A: M	***GetM-hrt(A)*** A: M				***SendData** for M* A: M *Update owner*

IM^D - Waiting for data

M, $M^T I$ - Valid data held by the core in modified state

Figure 7.2: Multiple pending requests from HRT and SRT cores.

At ❶, c_3^{srt} has a pending write request to A, which is serviced in the slack slot of c_0^{hrt}. Since there are no sharers for this cache line, the shared memory responds with A, and changes its state to M. On receiving the data ❷, the two timers for A are loaded with initial timeout values and begin to count down. At ❸, c_1^{hrt} does not have any pending request, and hence this *slack* slot is granted to c_2^{srt}, which broadcasts a write request to A. c_3^{srt} observes the write request from c_2^{srt} on the bus as *OtherGetM*-srt ❹, and changes its state from M to $M^T I$. It also marks the SRT core identifier c_2^{srt} in the `Dest-sl` field of A. In the next TDM slot of c_0^{hrt}, c_0^{hrt} has a pending write request to A that is serviced by the shared bus ❺. Now, c_2^{srt} observes a HRT core c_0^{hrt}'s write request as *OtherGetM*-hrt ❻, and reissues its write request. At the same time, c_3^{srt} also observes c_0^{hrt}'s write request ❼, and updates the `Dest-dd` and clears `Dest-sl` fields. This ensures that c_3^{srt} sends data to the HRT core c_0^{hrt} and not to SRT core c_2^{srt}. Since c_3^{srt} has A in $M^T I$ state and is waiting for its timer to expire, it does not respond to c_0^{hrt}'s request yet. At ❽, a read request from c_3^{srt} to A is a cache hit. Notice that this is a cache hit because of the timers, thereby providing performance improvement. In the absence of timers, the pending write requests to A would have resulted in A's invalidation in c_3^{srt} resulting in cache miss for this read request. At ❾, the timer $t_3(A, c_3^{\text{srt}}, c_0^{\text{hrt}})$ expires, and issues *SendData* message with requestor core

information set to c_0^{hrt}. c_3^{srt} moves from M^TI to M^IA state, indicating that it is waiting for the *Senddata* message to be ordered on the bus in the requesting core's slot (c_0^{hrt}). At c_0^{hrt}'s slot, which is the requesting core's slot, c_3^{srt}'s *SendData* message is broadcasted by the arbiter. c_3^{srt}, on observing its own *SendData* message ❿, sends the up-to-date A to c_0^{hrt}, and invalidates itself. The shared memory, on observing the *SendData* message, checks the pending message from PR LUT. Since the pending message is c_0^{hrt}'s write, the shared memory remains in M state and updates the owner to be c_0^{hrt}. c_0^{hrt} finally receives A sent by c_3^{srt}, completes its write operation ⓫, and moves to M state.

Multiple sharers

Figure 7.3 shows a scenario where there is a write request on a cache line A, that has multiple sharers. This example also assumes HourGlass(H-DD-WC-0), where SRT cores are not given any *dedicated* slots, and use only the *slack* slots. Initially assume that c_2^{srt} has a cache line A in shared (S) state, and it has obtained A in the *slack* slot of c_0^{hrt}. So, if the timer values are in terms of TDM periods, they expire at the end of c_0^{hrt}'s slot. The shared memory also has A in S state and have the sharer count as 1, indicating A has one sharer. The initial timeout value for all the timer configurations is 2 TDM periods.

At ❶, c_0^{hrt} broadcasts a read request on A and moves to the transient state IS^D, indicating that its request is broadcasted and that it is waiting for the data. Since HourGlass allows for the existence of multiple sharers, the shared memory responds with A to c_0^{hrt}, and updates the sharer count to 2. c_0^{hrt} receives A, moves to S state and begins the two timer values ❷. This is indicated in Figure 7.3 by the shaded region to show that the cache line is held by a core. At ❸, c_1^{hrt} broadcasts a write request on A. In order to maintain SWMR invariant, this write cannot proceed until the timers of the cache lines present in the two sharers expire. To keep track of this, the shared memory moves to a state SM ❹, showing that a core has made a write request on a cache line that is in shared state in atleast one core. c_0^{hrt} and c_2^{srt} both observe c_1^{hrt}'s write as $OtherGetM$-hrt and move to S^TI state and hold the cache line till their timers expire ❺. In the next slot of c_0^{hrt} ❻, the timer for A present in c_2^{srt}, $t_2(A, c_2^{\text{srt}}, c_1^{\text{hrt}})$, expires. As a result, c_2^{srt} issues *SelfInv* to be broadcasted at c_1^{hrt}'s slot and moves to S^IA state. At ❼, the *SelfInv* message of c_2^{srt} is broadcasted at the requesting core's slot, c_1^{hrt}. c_2^{srt}, on observing its own *SelfInv*, moves to a state SI, that denotes that its timer is expired, but there are still other sharers present. Hence any read on this cache line in this private cache is still valid and it is a hit. The shared memory, on observing a *SelfInv* message, decrements the sharer count by 1 ❽. It remains in SM state as the sharer count is non-zero indicating that there is at

TDM slots

CORES	c_0^{hrt}	c_1^{hrt}	c_0^{hrt}	c_1^{hrt}	c_0^{hrt}	c_1^{hrt}
c_0^{hrt}	Read A A: $I \rightarrow IS^D$ ❶ Recv A:50 ❷ A: $IS^D \rightarrow S$	❺ ***OtherGetM-hrt(A)*** A: $S \rightarrow S^T I$			$t_0(A, c_0^{hrt}, c_1^{hrt}) = 0$ Issue ***SelfInv*** ❾ A: $S^T I \rightarrow SI^A$	***Own_SelfInv*** A: $SI^A \rightarrow SI$ ***AllInv*** A: $SI \rightarrow I$
c_1^{hrt}		❸ Write A: 100 A: I → IM^D				Recv A: 100 ⓫ A: $IM^D \rightarrow M$
c_2^{srt}	***OtherGetS-hrt (A)*** A: S	❺ ***OtherGetM-hrt(A)*** A: $S \rightarrow S^T I$	$t_2(A, c_2^{srt}, c_1^{hrt}) = 0$ Issue ***SelfInv*** ❻ A: $S^T I \rightarrow SI^A$	❼ ***Own_SelfInv*** A: $SI^A \rightarrow SI$		***AllInv*** A: $SI \rightarrow I$
c_3^{srt}						
Shared memory	***GetS-hrt(A)*** A:S Count: 2 Send A:50	***GetM-hrt(A)*** A: $S \rightarrow SM$ Count: 2 ❹		***SelfInv(A)*** A: SM ❽ Count: 1		***SelfInv(A)*** Count: 0 ❿ A: $SM \rightarrow M$ ***Issue AllInv(A)*** Send A:50 to c_0^{hrt}

IM^D - Waiting for data

M, $M^T I$ - Valid data held by the core in modified state

IS^D - Waiting for data

S, $S^T I$ – Valid data held by the core in shared state

Figure 7.3: Multiple sharers.

least one sharer of A. At ❾, in c_0^{hrt}'s slot, A's timer in c_0^{hrt}, $t_0(A, c_0^{\mathsf{hrt}}, c_1^{\mathsf{hrt}})$, expires and *SelfInv* message is issued. This *SelfInv* message is broadcasted in c_1^{hrt}'s slot. The shared memory decrements the sharer count by 1 on seeing the *SelfInv* message. Once the sharer count is zero (all sharers have their timers expired), the shared memory issues a *AllInv* coherence message. Any core that has the cache line in SI state invalidates itself on seeing *AllInv* message, as all sharers have their timers expired. The shared memory then services the request present in the PR LUT. c_1^{hrt}'s write request is the pending request and hence, the shared memory sends data to c_1^{hrt} ❿. c_1^{hrt} finally receives A ⓫, completes its write operation and moves to M state.

Multiple pending requests in HourGlass(H-DD-WC)

Figure 7.4 illustrates a scenario where there are multiple pending requests from HRT and FRT cores when HourGlass(H-DD-WC) is used. This arbitration allocates all the HRT cores *dedicated* slots and few *dedicated* slots to FRT cores. This example assumes two HRT cores

and two FRT cores, with one *dedicated* slot allocated to FRT cores. Initially assume that the cache line A is not present in the private cache of any core. The initial value for timers is 2 TDM periods (here, 1 TDM period = 3 TDM slots).

Figure 7.4: Multiple pending requests in HourGlass(H-DD-WC).

At ❶, c_0^{hrt} requests for write operation on cache line A. The shared memory responds with A to c_0^{hrt} as A is not present in the private cache of any core. c_0^{hrt} receives A, completes its write operation and moves to M state and begins the count down of timers. c_0^{hrt} holds this copy of A for 2 TDM periods. At ❷, c_1^{hrt} does not have any pending request and hence this *slack* slot is utilized by FRT core c_2^{frt} to broadcast its write request on A. c_2^{frt} moves to a state IM^D, indicating that it is waiting for data. Since this request is broadcasted

in a *slack* slot, the arbiter provides this information along with the *GetM* message when it is broadcasted. So c_2^{frt} updates the bit Slot - `Sl` to *sl*, indicating that its request is broadcasted in a *slack* slot. c_1^{hrt} observes the write request as *OtherGetM*-frt ❸, and moves to M^TI state. It stores the requesting core's identifier in the destination field of the cache line. Since c_2^{frt}'s write was broadcasted in *slack* slot, `Dest-sl` is updated with c_2. The shared memory stores the pending request in the PR LUT shown as ❹ in Figure 7.4.

At ❺, in the *dedicated* slot of FRT cores, c_3^{frt} broadcasts a read request on A and moves to IS^D state. Since this is broadcasted in *dedicated* slot, the Slot bit is `dd`. c_2^{frt} observes this read request as *OtherGetS*-frt, and moves to a state IM^DI as pending FRT cores are services in FCFS manner ❻. It updates `Dest-dd` to c_3. c_0^{hrt} observes *OtherGetS*-frt and updates `Dest-dd` to c_3 as c_3^{frt}'s read request is broadcasted in *dedicated* slot. `Dest-dd` is updated only if it is the first request that is broadcasted in *dedicated* slot (i.e., `Dest-dd` is empty). The shared memory adds c_3^{frt}'s read request to PR LUT. At ❽, c_1^{hrt} broadcasts a write request on A and moves to IM^D state. On observing a HRT request, the FRT requests broadcasted in *slack* slots should be reissued. At ❾, c_2^{frt} observes *OtherGetM*-hrt and since its own request was broadcasted in *slack* slot (from `Sl` bit), c_2^{frt} reissues its write request and moves to IM^{AD} state. This reissue write request of c_2^{frt} is added to the RB (Reissue Buffer). c_0^{hrt}, on the other hand, observes c_1^{hrt}'s request and clears `Dest-sl` field as *slack* slot requests are reissued ❿. c_0^{hrt} now marks the first request broadcasted in *dedicated* slot as its destination, which is c_3^{frt}. Though c_3^{frt} observes a HRT request *OtherGetM*-hrt ⓫, it does not reissue its request as c_3^{frt} broadcasted its request in a *dedicated* slot. c_3^{frt} marks c_1 as its destination in `Dest-dd` and moves to IS^DI. The shared memory observes c_1^{hrt}'s write request and adds it to PR LUT ⓬. As discussed in Section 7.1.2, the shared memory cancels the pending FRT request from *slack* slot in PR LUT, c_2^{frt}'s write request, in the presence of a HRT request, c_1^{hrt}'s write request. At ⓭, in the *dedicated* slot of FRT cores, RB is checked first and hence c_2^{frt}'s write request is now broadcasted in *dedicated* slot. c_2^{frt} moves to IM^D state with Slot bit set to be `dd`. c_1^{hrt} and c_3^{frt} observe this request as *OtherGetM*-frt and make appropriate transitions ⓮. c_1^{hrt} moves to IM^DI and marks `Dest-dd` as c_2, indicating that once it receives data, it holds the data for a time duration and then sends A to c_2. c_3^{frt} remains in IS^DI state and does not update `Dest-dd` as it is non-empty. Shared memory adds c_2^{frt}'s write request to PR LUT.

At ⓯, the timer for A in c_0^{hrt}, $t_0(A, c_0^{\mathsf{hrt}}, c_3^{\mathsf{frt}})$, expires and c_0^{hrt} issues *SendData* for c_3^{frt} (from `Dest-dd`), and moves to M^IA state. *SendData* is broadcasted on the bus when the requesting core, c_3^{frt}, is assigned a slot. Assume that c_3^{frt} is allocated a slot only at ⓰, the *dedicated* slot for FRT cores. c_0^{hrt}, on observing its own *SendData*, invalidates

itself and sends data to c_3^{frt} and shared memory as c_3^{frt} requested A for read operation. The shared memory, on observing *SendData*, checks the PR LUT for pending message and moves to S^D state on seeing c_3^{frt}'s read request as the first pending request. When the shared memory receives A from c_0^{hrt}, it writes back the up-to-date value of A to the main memory and moves to SM state, indicating that a core has requested for write operation on A that is present in shared state in some core. The shared memory also increments the sharer count (not shown in the figure). Finally, c_3^{frt} receives A (17), and reads the up-to-date value 100 and moves to $S^T I$, indicating that there is a pending write request to A. Once c_3^{frt}'s timer expires, the shared memory sends A to c_1^{hrt}. After c_1^{hrt}'s request is serviced, c_2^{frt}'s write request is serviced (not shown in the figure).

Chapter 8

Timing Analysis

In this chapter, I derive the worst-case (WC) latency bound of a memory request, WCL_i^{cl}, that a core, c_i^{cl} where $cl \in \{\mathsf{hrt}, \mathsf{frt}, \mathsf{srt}\}$, suffers due to interference from other cores while accessing the shared memory. WCL_i^{cl} depends on the arbitration scheme chosen. WCL_i^{cl} for a core c_i (hrt, frt or srt), has three latency components: arbitration latency, access latency, and coherence latency.

Definition 1. ***Arbitration latency***, *L_i^{arb}, of any request generated by c_i is measured from the time stamp of its issuance until it is granted access to the bus. L_i^{arb} occurs due to prior requests from other cores scheduled before c_i.*

Definition 2. ***Access latency*** *is the time required to transfer the data requested by c_i from the shared memory or from the private cache of another core (cache-to-cache transfer) to the private cache of c_i. Recall that the TDM slot-width SW is set based on this latency and the transfer of necessary coherence messages required for data transfer to the requesting core. SW also accounts for the latency required to identify and replace a cache line based on the replacement policy when the cache is full. Hence, L^{acc} is given by SW, the upper bound on the latency required to transfer coherence messages and to transfer data from either the shared memory or from the private cache of another core.*

Definition 3. ***Coherence latency***, *L_i^{coh}, of any request generated by c_i is measured from the time stamp when the request is granted access to the bus until it starts its data transfer. L_i^{coh} is due to the coherence protocol rules that ensure data correctness in the cache hierarchy.*

Figure 8.1 illustrates the different latencies suffered by a core, c_0, when it issues a write request on a cache line A. At ❶, c_0 has a write request on A. However, this is not c_0's slot

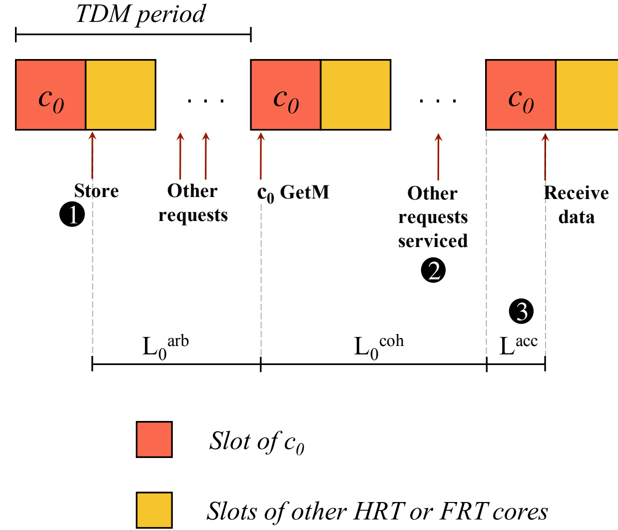


Figure 8.1: Different latency components per request of a core.

and hence the write request cannot be broadcasted on the bus. c_0 broadcasts the write request on bus with a coherence message *GetM* in its next slot. This latency accounts for the **arbitration latency** L_0^{arb}. Assume that requests by some other cores on A are broadcasted on the bus before c_0's request is broadcasted. In such a scenario, the other cores receive A and hold them for some time duration before c_0 receives it ❷. This latency, from the time *GetM* is broadcasted till c_0 is ready to receive data, accounts for **coherence latency** L_0^{coh}. Once c_0 is ready to receive data at its own slot, the shared memory sends data to c_0. This time required to transfer data to the private cache of c_0 from shared memory or from private cache of another core is given by **access latency** L^{acc} ❸. The total latency for a request issued by c_0 is the summation of L_0^{arb}, L_0^{coh} and L^{acc}.

With the addition of timers in HourGlass, any core, after its timer expires, sends a *SelfInv* message (if it is in shared state) or a *SendData* coherence message (if it is in modified state). These messages are broadcasted at the requesting core's slot. I represent this latency that is incurred due to transfer of data at requesting core's slot as $L_{req}^{cl'}$, where cl' is the criticality level of the requesting core. Figure 8.2 illustrates a case to calculate this latency. The example considers four cores with 2 HRT cores and 2 SRT cores, using H-DD-WC-0 arbitration, and hence the TDM period is 2 slots. Initially assume that c_2^{srt} obtained a cache line A in the *slack* slot of c_0^{hrt}. Then, in its next slot, c_0^{hrt} broadcasts write request on A. In the worst case, c_2^{srt}'s timer expires and sends *SelfInv* coherence message just after c_0^{hrt}'s start of slot. Hence, the message and data are transferred at c_0^{hrt}'s

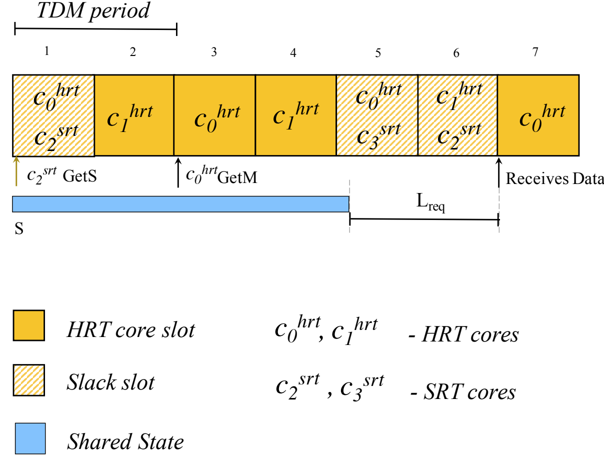


Figure 8.2: Illustration for the latency due to data transfer at requesting core slot.

next slot. As a result, the latency added due to data transfer at requesting core's slot for the HRT core, L_{req}^{hrt}, is given by the arbitration latency for the HRT core, in the worst case. This is 1 TDM period ($N_{\mathsf{hrt}} \times SW$), in this example. Similarly for a FRT core, L_{req}^{frt} is given by the arbitration latency for FRT core, in the worst case. Once the requesting core gets its slot, the coherence messages are broadcasted and data is transferred from the shared memory or from the private cache of some other core, and this is given by L^{acc}. After it receives the data, it starts its timer.

In this Chapter, I derive the WCL_i^{cl} for a core c_i based on the different arbitration schemes discussed in Chapter 5. Recall that N_{hrt}, N_{frt} and N_{srt} denote the number of HRT, FRT and SRT cores. N_s^{frt} denotes the number of *dedicated* slots allocated for FRT cores. N_s denotes the number of slots in the TDM schedule and is given by $N_s = N_{\mathsf{hrt}} + N_s^{\mathsf{frt}}$, as each HRT core is allocated one *dedicated* slot. TDM period is therefore $P = N_s \times SW$.

8.1 Timing Analysis for **HourGlass(H-DD-NWC)**

H-DD-NWC allocates N_{hrt} *dedicated* slots for HRT cores and N_s^{frt} *dedicated* slots for FRT cores, where $0 < N_s^{\mathsf{frt}} < N_{\mathsf{frt}}$. However, it is non-work-conserving and *slack* slots remain idle. Both HRT and FRT cores broadcast their requests on the bus only in their *dedicated* slots. They have a guaranteed worst-case latency bound per request.

8.1.1 Bound for HRT cores

The worst-case latency bound of a memory request that a HRT core, c_i^{hrt}, suffers is given by WCL_i^{hrt}. It is the summation of arbitration latency, coherence latency and the access latency.

Lemma 1. *The worst-case bus arbitration latency of a HRT core, c_i^{hrt}, $WCL_{bus,arb}^{hrt}$ occurs when the request is issued just after it missed c_i^{hrt}'s start of slot. Hence, in the worst case, it has to wait until its next slot, which is after 1 TDM period. $WCL_{bus,arb}^{hrt}$ for any c_i^{hrt} is given by:*

$$
\begin{aligned}
WCL_{bus,arb}^{hrt} &= N_s \times SW \\
&= (N_{hrt} + N_s^{frt}) \times SW
\end{aligned}
\tag{8.1}
$$

The WC coherence latency, $WCL_{i,coh}^{\mathsf{hrt}}$, for a request issued by c_i^{hrt} varies depending on the type of memory requests from HRT and FRT cores. Depending on whether the data is read-only or read-write, the WC coherence latency is affected. Further, sharing of data between HRT and FRT cores also affects the WC coherence latency. If data is not shared between HRT and FRT cores, then there is no interference between the two levels. Each core is affected by memory requests from other cores within the same criticality level. However, with data shared between HRT and FRT cores, the WC coherence latency of any core should now account for interference from both HRT and FRT cores.

Lemma 2. *For read-only data, the WC arbitration latency is equal to the WC bus arbitration latency, $WCL_{bus,arb}^{hrt}$, and the WC coherence latency is equal to zero, irrespective of whether it is shared or unshared between criticality levels as they are all read accesses.*

Read-Only Unshared:

$$WCL_{i,arb}^{hrt} = WCL_{bus,arb}^{hrt} \tag{8.2}$$

$$WCL_{i,coh}^{hrt} = 0 \tag{8.3}$$

Read-Only Shared:

$$WCL_{i,arb}^{hrt} = WCL_{bus,arb}^{hrt} \tag{8.4}$$

$$WCL_{i,coh}^{hrt} = 0 \tag{8.5}$$

Proof. If the access pattern of tasks is such that data is only read by HRT and FRT cores, then each core obtains all cache lines in shared state. Whenever a core c_i requests for a read operation on a cache line A, it takes bus arbitration latency (Lemma 1) to arrive at its slot. Since there are no write requests from a core, the WC arbitration latency for c_i^{hrt}, $WCL_{i,arb}^{\mathsf{hrt}}$, is given by the WC bus arbitration latency. In this slot of c_i, the shared memory sends A to c_i as this is read-only data and hence no cores have A in modified state. This accounts for the access latency, L^{acc}, for c_i to receive data. Thus, in case of read-only data, the cores receive data immediately once it reaches its own slot. There is no latency due to interference from other cores. Recall that the coherence latency of a request is measured from the time stamp c_i is granted access to the bus until it reaches the slot where it starts its data transfer. So, it does not include the latency where the actual data is transferred to c_i. This is given by access latency. Hence the WC coherence latency is zero for read-only data. This latency remains the same irrespective of whether data is shared or not between HRT and FRT cores as they are all read-only data and always obtain the cache line in shared state. □

Theorem 1. *When HRT and FRT cores request for read and write accesses on data, then WC total latency varies as there can be write requests to a cache line that is held in shared or modified state by any core. If data is not shared between HRT and FRT cores, then WC total latency of a core depends on the interference from other cores of the same criticality level. $WCL_{i,arb}^{hrt}$ for a request issued by c_i^{hrt} occurs under the critical instance when it requests for write on a cache line that c_i^{hrt} has just received in the shared state(S) in its private cache. $WCL_{i,coh}^{hrt}$ for a request issued by c_i^{hrt} occurs under the critical instance when all the remaining HRT cores issue write requests on the same cache line. For data that is requested for read and write operations by HRT cores, the worst case instance will be when all HRT cores request for write operation on the same cache line. Figure 8.3 illustrates the critical instance when $WCL_{i,arb}^{hrt}$ and $WCL_{i,coh}^{hrt}$ are observed for a HRT core, c_i^{hrt}.*

The critical instance is explained as follows:

1. *c_i^{hrt} issues a write request to a cache line A, which c_i^{hrt} has just received to be in the shared state (S) in its private cache and no other core has A in a valid state. Hence, the write request is broadcasted on the bus only after the timer for A, that holds A in shared state, in its own private cache expires.*

2. *During this time when c_i^{hrt} holds A in shared state and has not broadcasted its write on the bus, all the remaining ($N_{hrt} - 1$) HRT cores broadcast write requests to A before c_i^{hrt}'s write is broadcasted on the bus. Hence, by the arbitration policy, c_i^{hrt}'s write is serviced only after the write requests from HRT cores broadcasted before c_i^{hrt}*

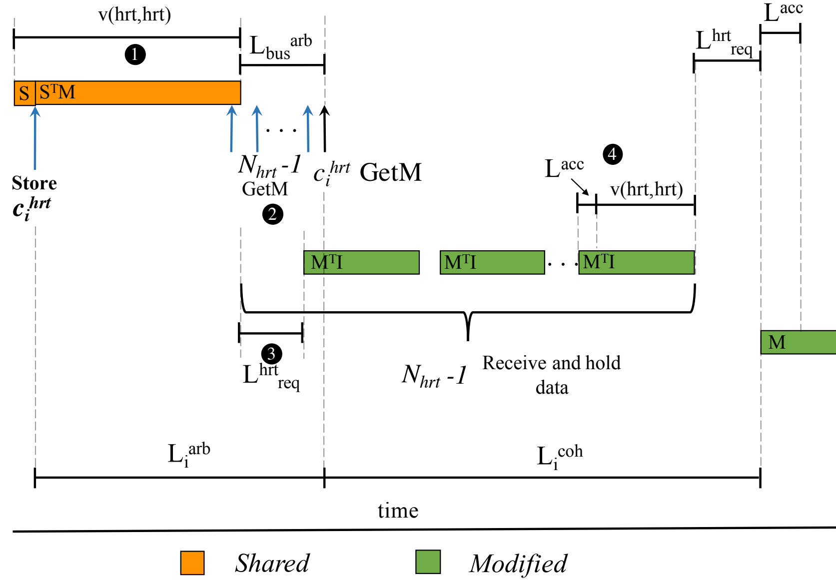


Figure 8.3: Critical Instance for H-DD-NWC HRT core - Read-Write Unshared.

are serviced. Each HRT core, after its write operation holds the cache line for a time duration and then invalidates.

The WC arbitration latency for a HRT core is given as:

Read-Write Unshared:

$$
\begin{aligned}
WCL_{i,arb}^{hrt} &= v(hrt, hrt) + WCL_{bus,arb}^{hrt} \\
&= v(hrt, hrt) + (N_s \times SW)
\end{aligned}
\tag{8.6}
$$

The WC coherence latency for a HRT core is given as:

Read-Write Unshared:

$$\begin{aligned}
WCL_{i,coh}^{hrt} =& (N_{hrt} - 1) \times \left(L_{req}^{hrt} + L^{acc} + v(hrt, hrt)\right) \\
&+ L_{req}^{hrt} \\
&- (N_s \times SW) \\
\\
=& (N_{hrt} - 1) \times \left((N_s \times SW) + SW + v(hrt, hrt)\right) \\
&+ (N_s \times SW) \\
&- (N_s \times SW) \\
\\
=& (N_{hrt} - 1) \times \left((N_s \times SW) + SW + v(hrt, hrt)\right)
\end{aligned} \tag{8.7}$$

The different components of equations 8.6 and 8.7 are explained as follows:

1. The first term of $WCL_{i,arb}^{\mathsf{hrt}}$, $v(\mathsf{hrt}, \mathsf{hrt})$, is the time duration c_i^{hrt} holds A in shared state, and c_i^{hrt}'s write should wait until it expires. This is shown as ❶ in Figure 8.3.

2. Only after the timer expires, c_i^{hrt} issues its write request on A, but this is broadcasted on the bus only at the start of c_i^{hrt}'s next slot (bus arbitration latency). In the worst case, the write request is issued just after c_i^{hrt} misses its start of slot. Hence, this will take a latency of $WCL_{bus,arb}^{i}$ given by the second term in equation 8.6.

3. Before c_i^{hrt} broadcasts its write on the bus, $(N_{\mathsf{hrt}} - 1)$ HRT cores broadcast write requests on A, ❷ in Figure 8.3. Each HRT core receives the data in its own slot and hence to arrive at this slot, I add the latency due to data transfer at the requesting core's slot, L_{req}^{hrt}, ❸ in Figure 8.3. Each HRT core incurs this L_{req}^{hrt} latency, which is given by the arbitration latency for a HRT core (as discussed initially in this Chapter). Each HRT core, after arriving at its slot, receives A after the transfer of necessary coherence messages and then completes its write operation. This accounts for L^{acc}, that is equal to 1 SW. Each HRT core, after its write operation, also holds A for a time duration $v(\mathsf{hrt}, \mathsf{hrt})$. These account for the first term in equation 8.7.

4. Finally the core under analysis, c_i^{hrt}, also incurs L_{req}^{hrt}, after the last HRT core has its timer for A expired shown as ❹ in Figure 8.3. This is given by the second term in the equation for $WCL_{i,coh}^{\mathsf{hrt}}$.

5. Recall that the coherence latency is measured from the time stamp c_i^{hrt}'s write is broadcasted on the bus. However, once the timer for c_i^{hrt} expires, the next requesting core receives the cache line, and this is shown as L_{req}^{hrt} of the requesting core ❸. Hence, part of the first term in equation 8.7 is not included in coherence latency. As discussed, once the timer for A in c_i^{hrt} expires, it takes bus arbitration latency $WCL_{bus,arb}^{\mathsf{hrt}}$ to broadcast its write request on the bus. Thus we subtract the additional part from first term in equation 8.7, which is equal to the bus arbitration latency.

Proof. I prove Theorem 1 by contradiction to show that the provided instance is the critical instance.

1. Suppose that the critical instance is when c_i^{hrt} issues write request on A which is not present in its cache (invalid state) or is present for a time duration $t > 0$ in shared state. Now, the first term of Equation 8.6 will either be removed or be replaced with $v(\mathsf{hrt}, \mathsf{hrt}) - t$. The observed arbitration latency is thus less than $WCL_{i,arb}^{\mathsf{hrt}}$ in equation 8.6 and so this is not the worst case arbitration latency. Hence, the critical instance must be when c_i^{hrt} requests for write on A immediately after it obtains A in shared state ($t = 0$).

2. Suppose that c_i^{hrt} requests for write on A when c_i^{hrt} is in shared state and there are H HRT cores also holding A in a valid (shared or modified) state, where $H \leq (N_{\mathsf{hrt}} - 1)$. Now all $N_{\mathsf{hrt}} - 1$ cannot make write requests to A before c_i^{hrt}'s write is broadcasted, as each of these H cores should either wait for their timers to time out (shared state) or are cache hits (modified). Hence, only N' HRT cores, where $N' = (N_{\mathsf{hrt}} - 1) - H$, broadcast write requests on A before c_i^{hrt} broadcasts its write request. The latency incurred due to N' HRT cores is less than the latency due to $(N_{\mathsf{hrt}} - 1)$ HRT cores, as $N' < (N_{\mathsf{hrt}} - 1)$. The observed coherence latency is thus less than $WCL_{i,coh}^{\mathsf{hrt}}$ in equation 8.7 and so this is not the critical instance. The critical instance should have pending requests from all the remaining $(N_{\mathsf{hrt}} - 1)$ HRT cores. Hence, the critical instance should be when no other core holds A in a valid state, when c_i^{hrt} requests for write on A.

3. Suppose that $v(\mathsf{hrt}, \mathsf{hrt})$ is large enough that a HRT core, c_k^{hrt} where $i \neq k$, receives more than 1 *dedicated* slot and so broadcasts multiple write requests on the cache line A. Now there are more than $(N_{\mathsf{hrt}} - 1)$ HRT pending requests before c_i^{hrt}'s write is broadcasted. However, in case of an in-order core, a core cannot issue another request to a same cache line A that already has a pending request by the same

core. Thus, by contradiction, $c_k^{\textsf{hrt}}$ can broadcast only one pending request per cache line A. $c_k^{\textsf{hrt}}$ can broadcast write request on another cache line B if there are more than 1 *dedicated* slot for $c_k^{\textsf{hrt}}$. However, this does not affect the coherence states corresponding to cache line A under analysis for $c_i^{\textsf{hrt}}$. A maximum of only $(N_{\textsf{hrt}} - 1)$ HRT cores can have pending requests for A before $c_i^{\textsf{hrt}}$'s write is broadcasted. Hence, the critical instance is when all $(N_{\textsf{hrt}} - 1)$ HRT cores broadcast write requests on A before $c_i^{\textsf{hrt}}$'s write is broadcasted.

4. Suppose that the critical instance is when the remaining HRT cores request for read on A instead of write. Since HourGlass allows for the presence of multiple sharers of the same cache line A, a core, c_i, requesting for a read on A, that is already present in shared state in some other core c_j, need not wait for c_j's timer to expire. In this scenario, the timer duration for which the cores hold A, overlap. This results in a latency less than $WCL_{i,coh}^{\textsf{hrt}}$ and hence this is not the critical instance. The critical instance should be when the remaining HRT cores request for write on A.

The above instances lead to either arbitration latency $L_i^{arb} < WCL_{i,arb}^{\textsf{hrt}}$ or coherence latency $L_i^{coh} < WCL_{i,coh}^{\textsf{hrt}}$. This proves by contradiction that the critical instance provided in Theorem 1 results in WC arbitration latency and WC coherence latency for a HRT core, thereby resulting in WC total latency per request for a HRT core. □

Theorem 2. *When data is shared between HRT and FRT cores and there are read and write accesses to the shared data from these cores, then WC arbitration latency,* $WCL_{i,arb}^{hrt}$*, and WC coherence latency,* $WCL_{i,coh}^{hrt}$*, depend on the interference from both HRT and FRT cores. The WC coherence latency,* $WCL_{i,coh}^{hrt}$*, for a request issued by* c_i^{hrt} *to a cache line A occurs under the critical instance where there is interference from all the remaining HRT and FRT cores. Figure 8.4 illustrates the critical instance when* $WCL_{i,coh}^{hrt}$ *is observed for a HRT core,* c_i^{hrt}*.*

The critical instance is explained as follows:

1. *Similar to the critical instance discussed in Theorem 1,* c_i^{hrt} *issues a write request to A, which* c_i^{hrt} *has just received to be in the shared state (S) in its private cache and no other core holds A in a valid state. Hence,* c_i^{hrt} *has to waits for its own timer for A to expire.*

2. *During this time, all the remaining* $(N_{hrt} - 1)$ *HRT cores broadcast write requests to A. Since, FRT cores are also allocated dedicated slots,* N_s^{frt} *FRT cores can broadcast*

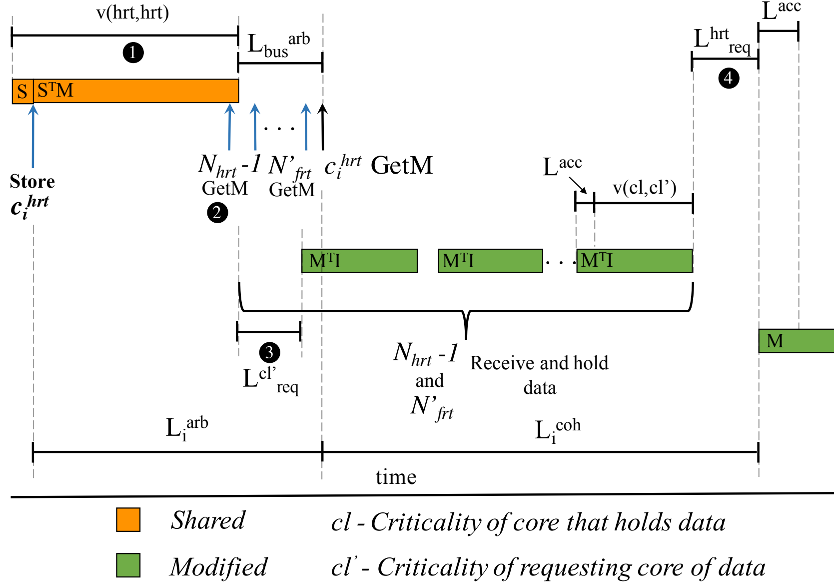


Figure 8.4: Critical Instance for H-DD-NWC HRT core - Read-Write Shared.

write requests per TDM period. Based on the initial value of timer $v(\mathsf{hrt}, \mathsf{hrt})$*, consider* N'_{frt} *FRT cores broadcast write requests to A before* c_i^{hrt}*'s write is broadcasted.*

3. c_i^{hrt}*'s write request is broadcasted only after these HRT and FRT requests. Hence, by the arbitration policy,* c_i^{hrt}*'s write is serviced only after the write requests from HRT and FRT cores broadcasted previously are serviced. Each HRT or FRT core, after its write operation holds the cache line for a time duration and then invalidates.*

The WC arbitration latency for a HRT core is given as:

***Read-Write Unshared**:*

$$\begin{aligned} WCL_{i,arb}^{\mathsf{hrt}} =& v(\mathsf{hrt}, \mathsf{hrt}) + WCL_{bus,arb}^{\mathsf{hrt}} \\ =& v(\mathsf{hrt}, \mathsf{hrt}) + (N_s \times SW) \end{aligned} \tag{8.8}$$

The WC coherence latency for a HRT core is given as:

Read-Write Shared:

$$
\begin{aligned}
WCL^{\mathsf{hrt}}_{i,coh} =& (N_{\mathsf{hrt}} - 1) \times \big(L^{\mathsf{hrt}}_{req} + L^{acc} + \max\{v(\mathsf{hrt},\mathsf{hrt}), v(\mathsf{hrt},\mathsf{frt})\}\big) \\
&+ (N'_{\mathsf{frt}}) \times \big(L^{\mathsf{frt}}_{req} + L^{acc} + \max\{v(\mathsf{frt},\mathsf{hrt}), v(\mathsf{frt},\mathsf{frt})\}\big) \\
&+ L^{\mathsf{hrt}}_{req} \\
&- (N_s \times SW) \\
\\
=& (N_{\mathsf{hrt}} - 1) \times \big((N_s \times SW) + SW + \max\{v(\mathsf{hrt},\mathsf{hrt}), v(\mathsf{hrt},\mathsf{frt})\}\big) \\
&+ \big(\min\{\left\lceil \frac{v(\mathsf{hrt},\mathsf{hrt}) + (N_s \times SW)}{N_s \times SW} \right\rceil \times N_s^{\mathsf{frt}}, N_{\mathsf{frt}}\}\big) \times \\
&\qquad\qquad ((\left\lceil \frac{N_{\mathsf{frt}}}{N_s^{\mathsf{frt}}} \right\rceil \times P) + SW + \max\{v(\mathsf{frt},\mathsf{hrt}), v(\mathsf{frt},\mathsf{frt})\}\big) \\
&+ (N_s \times SW) \\
&- (N_s \times SW) \\
\\
=& (N_{\mathsf{hrt}} - 1) \times \big((N_s \times SW) + SW + \max\{v(\mathsf{hrt},\mathsf{hrt}), v(\mathsf{hrt},\mathsf{frt})\}\big) \\
&+ \big(\min\{\left\lceil \frac{v(\mathsf{hrt},\mathsf{hrt}) + (N_s \times SW)}{N_s \times SW} \right\rceil \times N_s^{\mathsf{frt}}, N_{\mathsf{frt}}\}\big) \times \\
&\qquad\qquad ((\left\lceil \frac{N_{\mathsf{frt}}}{N_s^{\mathsf{frt}}} \right\rceil \times P) + SW + \max\{v(\mathsf{frt},\mathsf{hrt}), v(\mathsf{frt},\mathsf{frt})\}\big)
\end{aligned}
\tag{8.9}
$$

The different components of equations 8.8 and 8.9 are explained as follows:

1. The first term of equation 8.8, $v(\mathsf{hrt}, \mathsf{hrt})$, is the time duration c_i^{hrt} holds A in shared state, and c_i^{hrt}'s write should wait until it expires. This is shown as ❶ in Figure 8.4.

2. Only after the timer expires, c_i^{hrt} issues its write request on A, but this is broadcasted on the bus only at the start of c_i^{hrt}'s next slot (bus arbitration latency). In the worst case, c_i^{hrt}'s timer expires and the write request is issued just after c_i^{hrt} misses its start of slot. Hence, this will take a latency of $WCL^{\mathsf{hrt}}_{bus,arb}$, given by the second term in equation 8.8.

3. ❸ in Figure 8.4 shows the latency added due to data transfer at requesting core's slot. It is represented as $L_{req}^{cl'}$, where cl' represents the criticality of the requesting core. This latency is added for every HRT or FRT core that is required to be serviced.

4. Before c_i^{hrt}'s write is broadcasted on the bus, $(N_{\mathsf{hrt}} - 1)$ HRT cores broadcast write requests on A, ❷ in Figure 8.4. Each HRT core incurs the latency due to data transfer at requesting core's slot, L_{req}^{hrt}, that is given by 1 TDM period (arbitration latency), in the worst case, as HRT cores have a slot every TDM period. Each HRT core, after arriving at its slot, receives A after the transfer of necessary coherence messages and then completes its write operation. This accounts for L^{acc}, that is equal to 1 SW. Further, each HRT core, after its write operation, holds A for a time duration $v(\mathsf{hrt}, \mathsf{hrt})$ or $v(\mathsf{hrt}, \mathsf{frt})$ based on the criticality of the requesting core to which it has to send A. Hence, for the worst case scenario, I take the maximum of the two timer values.

5. The number of FRT cores that can broadcast write requests on A after c_i^{hrt} requests for write on A but before c_i^{hrt}'s write is broadcasted on the bus, N'_{frt}, depends on the duration $v(\mathsf{hrt}, \mathsf{hrt})$, arbitration latency $(N_s \times SW)$ and TDM period. This is because, in the worst case, only N_s^{frt} FRT cores can get access to the bus every TDM period. If $(v(\mathsf{hrt}, \mathsf{hrt}) + (N_s \times SW))$ is large (more than 1 TDM period), then in each TDM period, N_s^{frt} FRT cores broadcast write requests. So N'_{frt} is given as $\left\lceil \frac{v(\mathsf{hrt},\mathsf{hrt}) + (N_s \times SW)}{P} \right\rceil \times N_s^{\mathsf{frt}}$, where P is the TDM period. However if $(v(\mathsf{hrt}, \mathsf{hrt}) + (N_s \times SW))$ is very large, in the worst case, all N_{frt} FRT cores get access to the bus and broadcast write requests on A, before c_i^{hrt}'s write is broadcasted. This accounts for 'min' term in the second term of equation 8.9, that determines N'_{frt}. This is shown as N'_{frt} FRT requests ❷ in Figure 8.4. Since, H-DD-NWC allocates *dedicated* slots for FRT cores, these FRT cores are serviced before c_i^{hrt}. The second term of equation 8.9 includes the latency due to interference from each FRT core.

6. Each FRT core accounts for L_{req}^{frt}, latency due to data transfer at requesting core's slot, L^{acc}, latency to transfer coherence messages and receive data, and then $v(\mathsf{frt}, \mathsf{hrt})$ or $v(\mathsf{frt}, \mathsf{frt})$, the time duration it holds the cache line based on the criticality of the requesting core. L_{req}^{frt} for a FRT core is given by $\left\lceil \frac{N_{\mathsf{frt}}}{N_s^{\mathsf{frt}}} \right\rceil \times P$. This is the worst case when each FRT core just missed its slot in round-robin and hence it has to let all the other FRT cores to get access to the bus before it can get a slot.

7. Finally the core under analysis, c_i^{hrt}, also incurs L_{req}^{hrt}, after the last core (HRT or

FRT) has its timer for A expired shown as ❹ in Figure 8.4. This is given by the third term in the equation for $WCL^{\textsf{hrt}}_{i,coh}$.

8. Once the timer for A in $c^{\textsf{hrt}}_i$ expires, it takes arbitration latency $WCL^{\textsf{hrt}}_{i,arb}$ to broadcast its write request on the bus. By definition, coherence latency is measured from the time stamp the core is granted access to the bus, i.e., when $c^{\textsf{hrt}}_i$ broadcasts its write request. However, once the timer expires, the next requesting core receives the cache line, and this is included as $L^{cl'}_{req}$ of HRT or FRT requesting core ❸ and the time duration that core holds the cache line in equation 8.9. Thus a part of the first term in equation 8.9 should not be included in coherence latency. This is given by $WCL^{\textsf{hrt}}_{i,arb}$ shown as ❺ in Figure 8.4. I subtract the arbitration latency of $c^{\textsf{hrt}}_i$ from equation 8.9, which is 1 TDM period.

Proof. I prove Theorem 2 by contradiction to show that the provided instance is the critical instance.

1. Suppose that the critical instance is when $c^{\textsf{hrt}}_i$ requests for write on a cache line A that is not present in its cache (invalid state) or is present for a time duration $t > 0$ in shared state. As discussed in Theorem 1, now the first term of equation 8.8 will either be removed or be replaced with $v(\textsf{hrt}, \textsf{hrt}) - t$. The observed coherence latency is thus less than $WCL^{\textsf{hrt}}_{i,arb}$ in equation 8.8 and hence this is not the worst case latency. Hence, the critical instance must be when $c^{\textsf{hrt}}_i$ requests for write on A immediately after it obtains A in shared state ($t = 0$).

2. Suppose that the critical instance is when N' HRT cores, where $N' < (N_{\textsf{hrt}} - 1)$, broadcast write requests on A before $c^{\textsf{hrt}}_i$ broadcasts its write request. The latency incurred due to N' HRT cores is less than the latency due to $(N_{\textsf{hrt}} - 1)$ HRT cores, as $N' < (N_{\textsf{hrt}} - 1)$. The observed coherence latency is less than $WCL^{\textsf{hrt}}_{i,coh}$ in equation 8.9 and so this is not the worst case coherence latency. The critical instance should thus have pending write requests from all the remaining $(N_{\textsf{hrt}} - 1)$ HRT cores before $c^{\textsf{hrt}}_i$ broadcasts its write request. Hence, as discussed in Theorem 1, for $(N_{\textsf{hrt}} - 1)$ HRT cores to have pending write requests on A, initially no core holds A in a valid state.

3. Suppose that the critical instance is when N' FRT cores broadcast write requests on A before $c^{\textsf{hrt}}_i$ broadcasts its write request, where N' is less than the maximum number of FRT slots present between $c^{\textsf{hrt}}_i$'s write request and its issuance on the bus, $N'_{\textsf{frt}}$ (i.e., $N' < N'_{\textsf{frt}}$). Since $N' < N'_{\textsf{frt}}$, the latency incurred due to interference from N' FRT cores is less than the latency incurred due to $N'_{\textsf{frt}}$ FRT cores. This results in

coherence latency less than $WCL_{i,coh}^{\mathsf{hrt}}$ in equation 8.9 and hece this is not the qorst case latency. So this cannot be the critical instance.

4. Suppose that the critical instance is when a core c_k has more than 1 *dedicated* slot and hence broadcasts more than 1 write request for A before c_i^{hrt}'s broadcast of write request. Now there are more than $(N_{\mathsf{hrt}} - 1) + N'_{\mathsf{frt}}$ pending write requests from HRT and FRT cores. However, in case of an in-order core, a core cannot issue another request to a same cache line A that already has a pending request by the same core. As a result, if a core c_k has more than 1 *dedicated* slot, it can issue a read/write request to another cache line B before c_i^{hrt}'s broadcast of write request. This does not affect the worst latency for the request to A. Hence, at the maximum, there can be only $N_{\mathsf{hrt}} - 1$ pending HRT requests and N_{frt} pending FRT requests to the same cache line A before c_i^{hrt} broadcasts its write request as the critical instance.

5. Suppose that the critical instance is when HRT and FRT cores request for read on A instead of write. Since HourGlass allows for the presence of multiple sharers of the same cache line A, a core, c_i, requesting for a read on A, that is already present in shared state in some other core c_j, need not wait for c_j's timer to expire. In this scenario, the timer duration for which the cores hold A, overlap. This results in a latency less than $WCL_{i,coh}^{\mathsf{hrt}}$ and hence this is not the critical instance. Hence for the critical instance, all cores request for write operations on A.

The above instances lead to coherence latency $L_i^{coh} < WCL_{i,coh}^{\mathsf{hrt}}$ or arbitration latency $L_i^{arb} < WCL_{i,arb}^{\mathsf{hrt}}$. This proves by contradiction that the critical instance provided in Theorem 2 results in WC total latency for a HRT core. □

Theorem 3. *The total worst-case latency for a request issued by a HRT core c_i^{hrt} is given by the WC arbitration latency, WC coherence latency and the WC access latency for c_i^{hrt}. The WCL_i^{hrt} per request for c_i^{hrt} is given as follows:*

$$WCL_i^{hrt} = WCL_{i,arb}^{hrt} + WCL_{i,coh}^{hrt} + L^{acc} \tag{8.10}$$

8.1.2 Bound for FRT cores

I derive the worst-case latency bound of a memory request, WCL_i^{frt}, that a FRT core, c_i^{frt}, suffers due to interference from other cores. It is given by the summation of WC arbitration latency, WC coherence latency and access latency incurred by c_i^{frt}.

Lemma 3. *The worst-case bus arbitration latency of a* FRT *core,* c_i^{frt} *occurs when it has a write request on a cache line A just after* c_i^{frt} *has broadcasted a request on some other cache line B and all the remaining* $N_{\mathsf{frt}} - 1$ FRT *cores have a pending request to be broadcasted on the bus. Thus, WC bus arbitration latency* $WCL_{bus,arb}^{\mathsf{frt}}$ *for any* c_i^{frt} *is given by:*

$$\begin{aligned} WCL_{bus,arb}^{\mathsf{frt}} &= \left\lceil \frac{N_{\mathsf{frt}}}{N_s^{\mathsf{frt}}} \right\rceil \times P \\ &= \left\lceil \frac{N_{\mathsf{frt}}}{N_s^{\mathsf{frt}}} \right\rceil \times ((N_{\mathsf{hrt}} + N_s^{\mathsf{frt}}) \times SW) \end{aligned} \tag{8.11}$$

Proof. c_i^{frt} has a write request on A just after it has broadcasted a request to another cache line B. Now, there are also $N_{\mathsf{frt}} - 1$ FRT cores that are waiting to be granted access to the bus. Since the FRT cores are serviced in round-robin, c_i^{frt} can broadcast its request on A only after the remaining $(N_{\mathsf{frt}} - 1)$ FRT cores are granted access to the bus. In the worst case, all the $(N_{\mathsf{frt}} - 1)$ FRT cores have some pending request. So, c_i^{frt} is granted access to the bus only after $(N_{\mathsf{frt}} - 1)$ slots for FRT cores. However, only N_s^{frt} slots are allocated for FRT cores per TDM period. Hence, $WCL_{bus,arb}^{\mathsf{frt}}$ for c_i^{frt} is more than 1 TDM period and is given by Equation 8.11. □

The WC coherence latency, $WCL_{i,coh}^{\mathsf{frt}}$, for a request issued by c_i^{frt} depends on whether data is shared between HRT and FRT cores. As discussed previously, if data is not shared between HRT and FRT cores, then there is no interference between the two criticality levels. The WC latency of each core is affected by memory requests from other cores within the same criticality level. However, if data is shared between HRT and FRT cores, the WC coherence latency of any core should consider the interference from both HRT and FRT cores. WC coherence latency also depends on the type of memory requests from HRT and FRT cores. Depending on whether the data is read-only or read-write, the WC coherence latency varies.

Lemma 4. *For read-only data, the WC arbitration latency is equal to the WC bus arbitration latency and the WC coherence latency is zero for a* FRT *core, when data is shared or unshared between* HRT *and* FRT *cores.*

Read-Only Unshared:

$$WCL_{i,arb}^{\mathsf{frt}} = WCL_{bus,arb}^{\mathsf{frt}} \tag{8.12}$$

$$WCL_{i,coh}^{\mathsf{frt}} = 0 \tag{8.13}$$

Read-Only Shared:

$$WCL^{\mathsf{frt}}_{i,arb} = WCL^{\mathsf{frt}}_{bus,arb} \tag{8.14}$$

$$WCL^{\mathsf{frt}}_{i,coh} = 0 \tag{8.15}$$

Proof. If the access pattern of the tasks is such that the data is only read by the HRT and FRT cores, then all the cache lines exist in shared state in all the cores. As discussed in Section 8.1.1, proof of Lemma 2, a core receives data immediately once it reaches its own slot, when the data is read-only. Arbitration latency is given by the bus arbitration latency to arrive at its own slot. The WC coherence latency, on the other hand, is zero for read-only data. This latency remains the same irrespective of whether data is shared or not between HRT and FRT cores as they are all read-only data and always obtain the cache line in shared state. □

Theorem 4. *When FRT cores request for read and write accesses on data, then WC arbitration latency and WC coherence latency of a FRT core vary as there can be write requests to a cache line that is held in shared or modified state by some other core. However, if data is not shared between HRT and FRT cores, then WC coherence latency of a FRT core depends only on the interference from other FRT cores. $WCL^{\mathsf{frt}}_{i,arb}$ for a request issued by c^{frt}_i occurs under the critical instance when it requests for write on a cache line that c^{frt}_i has just received in the shared state(S) in its p rivate cache. $WCL^{\mathsf{frt}}_{i,coh}$ for a request issued by c^{frt}_i occurs under the critical instance when all the remaining FRT cores issue write requests on the same cache line. The worst case instance for a FRT core for read-write data will be when all FRT cores request for write operation on the same cache line. Figure 8.5 illustrates the critical instance when $WCL^{\mathsf{frt}}_{i,coh}$ is observed for a FRT core, c^{frt}_i.*

The critical instance is explained as follows:

1. *c^{frt}_i issues a write request to a cache line A, which c^{frt}_i has just received to be in the shared state (S) in its private cache and no other core holds A in a valid state. Now, the write request is broadcasted on the bus only after the timer for A, that holds A in shared state, in its own private cache expires.*

2. *During this time, all the remaining $(N_{\mathsf{frt}} - 1)$ FRT cores broadcast write requests to A before c^{frt}_i's write is broadcasted on the bus.*

Figure 8.5: Critical Instance for H-DD-NWC FRT core - Read-Write Unshared.

3. *c_i^{frt}'s write is serviced only after the write requests from FRT cores broadcasted before c_i^{frt} are serviced. Each FRT core, after its write operation holds the cache line for a time duration and then invalidates.*

The WC arbitration latency for a FRT core is given as:

***Read-Write Unshared**:*

$$\begin{aligned} WCL_{i,arb}^{frt} &= v(frt, frt) + WCL_{bus,arb}^{frt} \\ &= v(frt, frt) + \left(\left\lceil \frac{N_{frt}}{N_s^{frt}} \right\rceil \times P \right) \end{aligned} \tag{8.16}$$

The WC coherence latency for a request issued by a FRT core is as follows:

Read-Write Unshared:

$$
\begin{aligned}
WCL_{i,coh}^{\textsf{frt}} =& (N_{\textsf{frt}} - 1) \times \left(L_{req}^{\textsf{frt}} + L^{acc} + v(\textsf{frt}, \textsf{frt})\right) \\
& + L_{req}^{\textsf{frt}} \\
& - WCL_{bus,arb}^{\textsf{frt}} \\
\\
=& (N_{\textsf{frt}} - 1\}) \times \left(\left(\left\lceil \frac{N_{\textsf{frt}}}{N_s^{\textsf{frt}}} \right\rceil \times P\right) + SW + v(\textsf{frt}, \textsf{frt})\right) \\
& + \left(\left\lceil \frac{N_{\textsf{frt}}}{N_s^{\textsf{frt}}} \right\rceil \times P\right) \\
& - \left(\left\lceil \frac{N_{\textsf{frt}}}{N_s^{\textsf{frt}}} \right\rceil \times P\right) \\
\\
=& (N_{\textsf{frt}} - 1\}) \times \left(\left(\left\lceil \frac{N_{\textsf{frt}}}{N_s^{\textsf{frt}}} \right\rceil \times P\right) + SW + v(\textsf{frt}, \textsf{frt})\right) \qquad (8.17)
\end{aligned}
$$

The different components of equations 8.16 and 8.17 are explained as follows:

1. The WC arbitration latency $WCL_{i,arb}^{\textsf{frt}}$ is derived in a similar way compared to $WCL_{i,arb}^{\textsf{hrt}}$ in Section 8.1.1. Since $c_i^{\textsf{frt}}$ holds A in shared state for $v(\textsf{frt}, \textsf{frt})$ before its write can be broadcasted, this latency is included in $WCL_{i,arb}^{\textsf{frt}}$ (first term), represented as ❶ in Figure 8.5. It takes bus arbitration latency to broadcast this write request on the bus, given by $WCL_{bus,arb}^{\textsf{frt}}$ (second term), in the worst case.

2. The first term of equation 8.17 accounts for the latency due to interference from the remaining FRT cores. WC arbitration latency for $c_i^{\textsf{frt}}$ allows for all remaining FRT cores to be granted access to the bus before $c_i^{\textsf{frt}}$ is granted access. Hence, in the worst case, all the remaining $(N_{\textsf{frt}} - 1)$ FRT cores broadcast write requests to A before $c_i^{\textsf{frt}}$ broadcasts its write request (shown as ❷ in Figure 8.5). Each FRT core incurs $L_{req}^{\textsf{frt}}$, latency due to data transfer at requesting core's slot, L^{acc}, latency to transfer data from the shared memory or from the private cache of any core, and the latency due to timers as each core holds A in valid state for some time duration ($v(\textsf{frt}, \textsf{frt})$.

3. The second term in the equation 8.17 accounts for the latency due to data transfer at requesting core's slot (L^{frt}_{req}) for c^{frt}_i, the FRT core under analysis. This is represented as ❸ in Figure 8.5.

4. Recall that the coherence latency is measured from the time stamp the c^{frt}_i's write is broadcasted on the bus. However, once the timer for c^{frt}_i expires, the next requesting core receives the cache line, and this accounts for L^{frt}_{req} and $v(\mathsf{frt},\mathsf{frt}$ of the next requesting core ❹. Hence, part of the first term in equation 8.17 is not included in coherence latency. As discussed, once the timer for A in c^{frt}_i expires, it takes bus arbitration latency $WCL^{\mathsf{frt}}_{bus,arb}$ to broadcast its write request on the bus. Thus we subtract the additional part from first term in equation 8.17, which is equal to the bus arbitration latency.

Proof. The proof for Theorem 4 is by contradiction. It shows that the instance discussed is the critical instance that results in WC coherence latency.

1. Suppose that c^{frt}_i requests for write on A which is not present in its private cache (invalid state) or which is present in c^{frt}_i in shared state for a time duration t, where $0 < t < v(\mathsf{frt},\mathsf{frt})$. Now the first term $v(\mathsf{frt},\mathsf{frt})$ is either not included in $WCL^{\mathsf{frt}}_{i,arb}$ or is replaced with $v(\mathsf{frt},\mathsf{frt}) - t$. This results results in arbitration latency less that the latency presented in equation 8.16. So the critical instance must be when c^{frt}_i has a write request on A immediately after it receives A in shared state.

2. Suppose that the critical instance is when N' FRT cores broadcast write requests on A before c^{frt}_i broadcasts its write request, where $N' < N_{\mathsf{frt}} - 1$. The latency due to interference from FRT cores is given by the first term in equation 8.17. Since $N' < N_{\mathsf{frt}} - 1$, the latency due to interference from N' FRT cores is less than the latency due to interference from $N_{\mathsf{frt}} - 1$ FRT cores. This results in the observed coherence latency L^{coh}_i that is less than $WCL^{\mathsf{frt}}_{i,coh}$ provided in 8.17. Hence, as discussed in Theorem 1, the critical instance should be when all the remaining FRT cores request for A only after c^{frt}_i's write request.

3. Supppose that the critical instance is when a core c_k broadcasts more than 1 write request on A, resulting in more than $N_{\mathsf{frt}} - 1$ pending FRT requests before c^{frt}_i's write request. In case of an in-order core, a core cannot issue another request to a same cache line A that already has a pending request by the same core. As a result, if a core c_k has more than 1 *dedicated* slot, it can issue a read/write request to another cache line B before c^{frt}_i's broadcast of write request, but not A. Hence,

at the maximum, there can be only $N_{\textsf{frt}}-1$ pending FRT requests to the same cache line A before $c_i^{\textsf{frt}}$ broadcasts its write request that provides the worst case coherence latency.

4. Suppose that the critical instance is when FRT cores broadcast read requests instead of write requests on A. In this case, multiple copies of A exist in the private cache of different cores in shared state, as HourGlass allows for the existence of multiple sharers of a cache line at a time. As a result, the latency due to the interference from each core, holding A for some time duration, overlap. This results in a latency less than $WCL_{i,coh}^{\textsf{frt}}$ and hence this is not the worst case coherence latency. The critical instance is when $N_{\textsf{frt}}-1$ FRT cores have pending write requests to A before $c_i^{\textsf{frt}}$ broadcasts its write request.

The instances discussed above lead to either arbitration latency $L_i^{arb} < WCL_{i,arb}^{\textsf{frt}}$ or coherence latency $L_i^{coh} < WCL_{i,coh}^{\textsf{frt}}$. This proves by contradiction that the critical instance provided in Theorem 4 results in WC arbitration and WC coherence latency, thus WC total latency for a FRT core. □

Theorem 5. *When data is shared between HRT and FRT cores, then the WC coherence latency should account for the interference from both HRT and FRT cores. WC arbitration latency, $WCL_{i,arb}^{\textsf{frt}}$, for a request issued by $c_i^{\textsf{frt}}$ occurs under the critical instance when it requests for write on a cache line that $c_i^{\textsf{frt}}$ has just received in the shared state(S) in its private cache. The WC coherence latency, $WCL_{i,coh}^{\textsf{frt}}$, for a request issued by $c_i^{\textsf{frt}}$ to a cache line A occurs under the critical instance where there is interference from all the remaining HRT and FRT cores. Figure 8.6 shows the critical instance when $WCL_{i,arb}^{\textsf{frt}}$ and $WCL_{i,coh}^{\textsf{frt}}$ are observed for a FRT core, $c_i^{\textsf{frt}}$.*

The critical instance where $WCL_{i,coh}^{\textsf{frt}}$ is obtained for FRT core, $c_i^{\textsf{frt}}$, is explained as follows:

1. *$c_i^{\textsf{frt}}$ requests for write on a cache line A, which it has just received in shared state in its private cache and no other core holds A in a valid state. Now, $c_i^{\textsf{frt}}$ has to wait for A's timer in its own cache to expire before it can broadcast the write request on the bus.*

2. *In this time duration, all the remaining HRT ($N_{\textsf{hrt}}$) and FRT ($N_{\textsf{frt}}-1$) cores broadcast write requests to A before $c_i^{\textsf{frt}}$'s write is broadcasted.*

Figure 8.6: Critical Instance for H-DD-NWC FRT core - Read-Write Shared.

3. *c_i^{frt}'s write is serviced only after the* HRT *and* FRT *cores, that broadcasted requests before c_i^{frt} write is broadcasted, are serviced. Each of this core holds the cache line for some time duration after its write operation and invalidates.*

The WC arbitration latency for a FRT *core is given as:*

***Read-Write Shared**:*

$$
\begin{aligned}
WCL_{i,arb}^{\mathsf{frt}} =& v(\mathsf{frt}, \mathsf{frt}) + WCL_{bus,arb}^{\mathsf{frt}} \\
=& v(\mathsf{frt}, \mathsf{frt}) + (\left\lceil \frac{N_{\mathsf{frt}}}{N_s^{\mathsf{frt}}} \right\rceil \times P)
\end{aligned} \quad (8.18)
$$

The WC coherence latency for a request issued by a FRT *core is as follows:*

Read-Write Shared:

$$
\begin{aligned}
WCL_{i,coh}^{\textit{frt}} = & N_{\textit{hrt}} \times \big(L_{req}^{\textit{hrt}} + L^{acc} + \max\{v(\textit{hrt},\textit{hrt}), v(\textit{hrt},\textit{frt})\}\big) \\
& + (N_{\textit{frt}} - 1) \times \big(L_{req}^{\textit{frt}} + L^{acc} + \max\{v(\textit{frt},\textit{hrt}), v(\textit{frt},\textit{frt})\}\big) \\
& + L_{req}^{\textit{frt}} \\
& - WCL_{bus,arb}^{\textit{frt}}
\end{aligned}
$$

$$
\begin{aligned}
WCL_{i,coh}^{\textit{frt}} = & N_{\textit{hrt}} \times \big((N_s \times SW) + SW + \max\{v(\textit{hrt},\textit{hrt}), v(\textit{hrt},\textit{frt})\}\big) \\
& + (N_{\textit{frt}} - 1\}) \times \Big((\Big\lceil \frac{N_{\textit{frt}}}{N_s^{\textit{frt}}} \Big\rceil \times P) + SW + \max\{v(\textit{frt},\textit{hrt}), v(\textit{frt},\textit{frt})\}\Big) \\
& + (\Big\lceil \frac{N_{\textit{frt}}}{N_s^{\textit{frt}}} \Big\rceil \times P) \\
& - (\Big\lceil \frac{N_{\textit{frt}}}{N_s^{\textit{frt}}} \Big\rceil \times P) \\
\\
= & (N_{\textit{hrt}}) \times \big((N_s \times SW) + SW + \max\{v(\textit{hrt},\textit{hrt}), v(\textit{hrt},\textit{frt})\}\big) \\
& + (N_{\textit{frt}} - 1) \times \Big((\Big\lceil \frac{N_{\textit{frt}}}{N_s^{\textit{frt}}} \Big\rceil \times P) + SW + \max\{v(\textit{frt},\textit{hrt}), v(\textit{frt},\textit{frt})\}\Big)
\end{aligned} \tag{8.19}
$$

The different components of equations 8.18 and 8.19 are explained as follows:

1. Since c_i^{frt} holds A in shared state for $v(\mathsf{frt},\mathsf{frt})$ before its write can be broadcasted, this latency is included in $WCL_{i,arb}^{\mathsf{frt}}$ (first term), represented as ❶ in Figure 8.6. After c_i^{frt}'s timer expires, the write is broadcasted in c_i^{frt}'s slot which takes bus arbitration latency, $WCL_{bus,arb}^{\mathsf{frt}}$, in the worst case.

2. All the HRT cores broadcast write requests to A (❷ in Figure 8.6). Each of this HRT core incurs the latency L_{req}^{hrt} due to data transfer at requesting core's slot, data access latency, L^{acc}, and the latency due to the core holding valid A in its cache for a fixed time duration (depends on the criticality of the requesting core). As discussed previously, L_{req}^{hrt} is given by 1 TDM period (arbitration latency for HRT cores). This accounts for the first term in equation 8.19.

3. The second term accounts for the latency due to interference from FRT cores. WC arbitration latency for c_i^{frt} allows for all remaining FRT cores to be granted access to the bus before c_i^{frt} is granted access. Hence, in the worst case, all the remaining $(N_{\mathsf{frt}} - 1)$ FRT cores broadcast write requests to A before c_i^{srt} broadcasts its write request (shown as ❷ in Figure 8.6). Each FRT core incurs latency due to timers (holds A in valid state), access latency, L^{acc}, and latency due to data transfer at requesting core's slot L_{req}^{frt}.

4. The third term in the equation accounts for the latency due to data transfer at requesting core's slot (L_{req}^{frt}) for c_i^{frt}, the FRT core under analysis. This is represented as ❸ in Figure 8.6.

5. Recall that the coherence latency is measured from the time stamp the c_i^{frt}'s write is broadcasted on the bus. However, once the timer for c_i^{frt} expires, the next requesting core receives the cache line, and this accounts for $L_{req}^{cl'}$ and $v(\mathsf{frt}, cl')$ of the next requesting core, $c_i^{cl'}$, ❹. Hence, part of the first term in equation 8.17 is not included in coherence latency. As discussed, once the timer for A in c_i^{frt} expires, it takes bus arbitration latency $WCL_{bus,arb}^{\mathsf{frt}}$ to broadcast its write request on the bus. Thus we subtract the additional part from first term in equation 8.17, which is equal to the bus arbitration latency.

Proof. The proof for Theorem 5 is by contradiction. It shows that the instance discussed is the critical instance that results in WC coherence latency.

1. Suppose that the critical instance is when c_i^{frt} requests for write on A which is not present in its private cache (invalid state) or which is present in c_i^{frt} in shared state for a time duration t, where $0 < t < v(\mathsf{frt}, \mathsf{frt})$. Now the first term $v(\mathsf{frt}, \mathsf{frt})$ is either not included in $WCL_{i,arb}^{\mathsf{frt}}$ or is replaced with $v(\mathsf{frt}, \mathsf{frt}) - t$. This results results in arbitration latency less that the latency presented in equation 8.18 and hence this instance will not result in the worst case arbitration latency. So the critical instance must be when c_i^{frt} has a write request on A immediately after it receives A in shared state.

2. Suppose that the critical instance be when N' HRT cores broadcast write requests on A before c_i^{frt}'s write request is broadcasted, where $N' < N_{\mathsf{hrt}}$. Now the observed coherence latency includes the interference from N' HRT cores given as $N' \times \big(\max\{v(\mathsf{hrt}, \mathsf{hrt}),\, v(\mathsf{hrt}, \mathsf{frt})\} + L_{req}^{\mathsf{hrt}}\big)$. However this term is less than the first term in equation 8.19, as $N' < N_{\mathsf{hrt}}$, resulting in a latency less than $WCL_{i,coh}^{\mathsf{frt}}$. Hence this is

not the critical instance. Critical instance is when $N' < N_{\mathsf{hrt}}$ cores broadcast write requests before c_i^{frt}'s write request is broadcasted.

3. Suppose that the critical instance is when N' FRT cores broadcast write requests on A before c_i^{frt} broadcasts its write request, where $N' < N_{\mathsf{frt}} - 1$. The latency due to interference from FRT cores is given by the second term in equation 8.19. Since $N' < N_{\mathsf{frt}} - 1$, the latency due to interference from N' FRT cores is less than the latency due to interference from $N_{\mathsf{frt}} - 1$ FRT cores. This results in the observed coherence latency L_i^{coh} that is less than $WCL_{i,coh}^{\mathsf{frt}}$ provided in 8.19 and hence this is not the critical instance. Critical instance is when $N_{\mathsf{frt}} - 1$ cores broadcast write requests before c_i^{frt}'s write request is broadcasted.

4. Suppose that the critical instance is when c_i^{frt} requests for write on A, atleast one core, c_k, has A in a valid state (shared or modified). If c_k is in shared state, then it can broadcast its request only after its own timer and c_i^{frt}'s timer expire. Based on arbitration, there can be a case when c_i^{frt}'s write is broadcasted before c_k. If c_k is in modified state, then it is a cache hit and so there is no need of broadcast request to A. Thus, this results in pending write requests either less than N_{hrt} HRT cores or $N_{\mathsf{frt}} - 1$ FRT cores. But as discussed before, critical instance is when N_{hrt} HRT cores and $N_{\mathsf{frt}} - 1$ FRT cores have pending write requests. Hence, critical instance is when no other core holds A in a valid state when c_i^{frt} requests for write on A.

5. Supppose that the critical instance is when a core c_k broadcasts more than 1 write request on A, resulting in more than $N_{\mathsf{frt}} - 1$ pending FRT requests or more than N_{hrt} HRT requests before c_i^{frt}'s write request is broadcasted. However, in case of an in-order core, a core cannot issue another request to a same cache line A that already has a pending request by the same core. As a result, if a core c_k has more than 1 *dedicated* slot, it can issue a read/write request to another cache line B before c_i^{frt}'s broadcast of write request, but not A. Hence, at the maximum, there can be only N_{hrt} pending HRT requests and $N_{\mathsf{frt}} - 1$ pending FRT requests to the same cache line A before c_i^{frt} broadcasts its write request.

6. Suppose that the critical instance is when HRT and FRT cores broadcast read requests instead of write requests on A. In this case, multiple copies of A exist in the private cache of different cores in shared state, as HourGlass allows for the existence of multiple sharers of a cache line at a time. As a result, the latency due to the interference from each core, holding A for some time duration, overlap. This results in a latency less

than $WCL^{\mathsf{frt}}_{i,coh}$ and hence this is not lead to worst-case coherence latency. Hence, the critical instance is when all the other cores request for write on A.

The instances discussed above lead to arbitration latency $L^{arb}_i < WCL^{\mathsf{frt}}_{i,arb}$ or coherence latency $L^{coh}_i < WCL^{\mathsf{frt}}_{i,coh}$. This proves by contradiction that the critical instance provided in Theorem 5 results in WC total latency for a FRT core. □

Theorem 6. *The total worst-case latency for a request issued by a* FRT *core* c^{frt}_i *is given by the WC arbitration latency, WC coherence latency and the WC access latency for* c^{frt}_i*. The* WCL^{frt}_i *per request for* c^{frt}_i *is given as follows:*

$$WCL^{\mathsf{frt}}_i = WCL^{\mathsf{frt}}_{i,arb} + WCL^{\mathsf{frt}}_{i,coh} + L^{acc} \tag{8.20}$$

8.2 Timing Analysis for HourGlass(H-DD-WC)

H-DD-WC is similar to H-DD-NWC but is work-conserving. The *slack* slots of HRT cores are utilized by FRT or SRT cores to broadcast pending requests. Since HRT and FRT cores have *dedicated* slots, they have a guaranteed WC latency bound. For the special case of this arbitration where slots are allocated only for HRT cores, H-DD-WC-0, the cores of the second criticality level, SRT, do not have a guaranteed bound.

8.2.1 Bound for HRT cores

The only difference of this arbitration from H-DD-NWC is that the *slack* slots of HRT cores are utilized by FRT or SRT cores. However, requests from FRT cores (in case of H-DD-WC) or from SRT cores (in case of H-DD-WC-0) broadcasted in *slack* slots are reissued in the presence of a request from HRT core. If the FRT cores broadcast requests in *dedicated* slots, then a HRT core c^{hrt}_i is serviced only after all these FRT cores are serviced. Thus, the HRT cores are still guaranteed a WCL bound as the number of slots allocated for FRT cores is fixed, N^{frt}_s, and the requests broadcasted in *slack* slots are reissued. The analysis for WCL^{hrt}_i for a HRT core when H-DD-WC is used is similar to the WC latency discussed in Section 8.1 and is given by equation 8.10.

For H-DD-WC which has N^{frt}_s as non-zero, for all different cases, where data is shared or not between different criticality levels or when the memory accesses are read-only or

read-write, the analysis for WC arbitration latency and WC coherence latency is similar to the analysis for H-DD-NWC in Lemma 2 and Theorems 1 and 2.

However, for the special case of arbitration H-DD-WC-0 where $N_s^{\mathsf{frt}} = 0$, the worst case instance for the case where data is read-write and also shared between different criticality levels, is different compared to H-DD-NWC and H-DD-WC. For the other scenarios, such as **Read-Only Unshared, Read-Only Shared, Read-Write Unshared**, the analysis for WC arbitration latency and WC coherence latency and hence the WC total latency is the same as that of H-DD-NWC with N_s^{frt} set as 0. Here, I discuss the WC arbitration latency and WC coherence latency for a HRT core, with read-write data shared between HRT and SRT cores, when H-DD-WC-0 is used. The WC bus arbitration latency, on the other hand, is given by Lemma 1. By substituting $N_s^{\mathsf{frt}} = 0$ in equation 8.1, it is observed that the WC bus arbitration latency is tighter compared to H-DD-NWC and H-DD-WC.

Theorem 7. *When H-DD-WC-0 is used, WC arbitration latency, $WCL_{i,arb}^{\mathsf{hrt}}$, and WC coherence latency, $WCL_{i,coh}^{\mathsf{hrt}}$, for a request issued by c_i^{hrt} to a cache line A, that is shared between HRT and SRT cores and is read-write, occurs under the critical instance, where there is interference from one SRT core and all the remaining HRT cores. Figure 8.7 illustrates the critical instance under which the WC total latency for a HRT request is observed.*

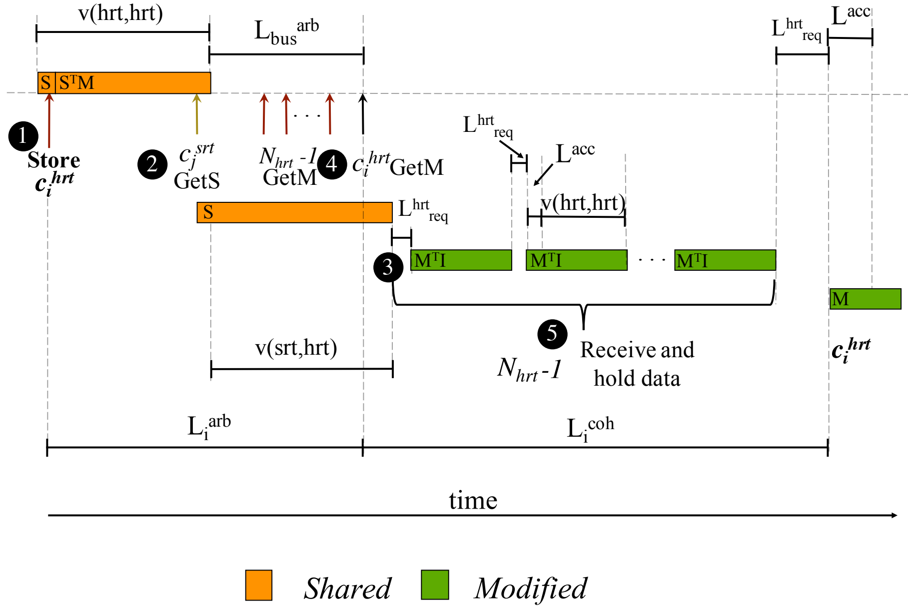


Figure 8.7: Critical Instance for H-DD-WC-0 HRT core - Read-Write Shared.

The critical instance is explained as follows:

1. c_i^{hrt} *issues a write request to A, which* c_i^{hrt} *has just received to be in the shared state (S) in its private cache and begins its timer for A and no other core holds A in valid state. The write request is issued only after the timer for A, that holds A in shared state, in its own private cache expires.*
2. *When* c_i^{hrt}*'s timer is about to expire, a SRT core* c_j^{srt} *requests for a read on A and obtains it in the shared state, as HourGlass allows for the presence of multiple sharers of a cache line. Thus, there are now two sharers of A.* c_j^{srt} *now holds A for a fixed time duration* $v(srt, hrt)$*. The interference from SRT core is dependent on the values of* $v(srt, hrt)$ *and* $v(hrt, hrt)$*.*
3. *After* c_j^{srt} *obtains A in shared state, but before* c_i^{hrt}*'s write is broadcasted on the bus, all the other* $(N_{hrt} - 1)$ *HRT cores broadcast write requests to A. The write request to any core can be serviced only after the timer of A expires for the last sharer (either* c_i^{hrt} *or* c_j^{srt}*).*
4. *Since* c_i^{hrt}*'s write is broadcasted only after* $N_{hrt} - 1$ *write requests, by first-come first-serve policy,* c_i^{hrt}*'s write is serviced only after the* $N_{hrt} - 1$ *write requests from HRT cores are serviced. Each HRT core, after its write operation, holds the data for a time duration* $v(hrt, hrt)$ *and then invalidates.*

The WC arbitration latency for a HRT core for H-DD-WC-0, where $N_s^{frt} = 0$*, is given by:*

Read-Write Shared*:*

$$\begin{aligned} WCL_{i,arb}^{hrt} &= v(hrt, hrt) + WCL_{bus,arb}^{hrt} \\ &= v(hrt, hrt) + \left(N_{hrt} \times SW\right) \end{aligned} \tag{8.21}$$

Consider a term X, that is based on the initial values of timer configurations $v(srt, hrt)$ *and* $v(hrt, hrt)$ *compared to the TDM period. X is given by,*

$$X = \begin{cases} 0, & \text{if } v(srt, hrt) \text{or } v(hrt, hrt) < (N_{hrt} \times SW) \\ 1, & \text{otherwise} \end{cases}$$

The WC coherence latency is given as:

Read-Write Unshared:

$$
\begin{aligned}
WCL^{\mathsf{hrt}}_{i,coh} =& \big(X \times v(\mathsf{srt}, \mathsf{hrt})\big) \\
& + (N_{\mathsf{hrt}} - 1) \times (L^{\mathsf{hrt}}_{req} + L^{acc} + v(\mathsf{hrt}, \mathsf{hrt})) \\
& + L^{\mathsf{hrt}}_{req} \\
& - WCL^{\mathsf{hrt}}_{bus,arb}
\end{aligned}
$$

$$
\begin{aligned}
WCL^{\mathsf{hrt}}_{i,coh} =& \big(X \times v(\mathsf{srt}, \mathsf{hrt})\big) \\
& + (N_{\mathsf{hrt}} - 1) \times \big((N_{\mathsf{hrt}} \times SW) + SW + v(\mathsf{hrt}, \mathsf{hrt})\big) \\
& + (N_{\mathsf{hrt}} \times SW) \\
& - (N_{\mathsf{hrt}} \times SW)
\end{aligned}
$$

$$
\begin{aligned}
WCL^{\mathsf{hrt}}_{i,coh} =& \big(X \times v(\mathsf{srt}, \mathsf{hrt})\big) \\
& + (N_{\mathsf{hrt}} - 1) \times \big((N_{\mathsf{hrt}} \times SW) + SW + v(\mathsf{hrt}, \mathsf{hrt})\big)
\end{aligned}
\tag{8.22}
$$

The different components in the above equations 8.21 and 8.22 is explained with the help of Figure 8.7.

1. c^{hrt}_i issues a write request to A present in shared state ❶ and so it has to wait for its own timer to expire. Since it requested for A immediately after it obtained A in shared state, it has to wait for $v(\mathsf{hrt}, \mathsf{hrt})$, the first term in equation 8.21.

2. After c^{hrt}_i's timer expires, c^{hrt}_i issues its write request on A, that is broadcasted on the bus in c^{hrt}_i's slot. This accounts for the bus arbitration latency. In the worst case, it just missed its start of slot and thus this is given by the WC bus arbitration latency. This is given by the second term in equation 8.21.

3. After the c^{srt}_j obtains A in shared state ❷, there are two sharers of A: c^{hrt}_i and c^{srt}_j. Any write request to A by any other core can be serviced only after A's timer expires in the private cache of the last sharer. Hence the term $v(\mathsf{srt}, \mathsf{hrt})$, first term of equation 8.22 is added as c^{srt}_j is the last sharer.

4. However, the interference from SRT component, $v(\mathsf{srt}, \mathsf{hrt})$ (first term), is added depending on the value of $v(\mathsf{srt}, \mathsf{hrt})$, $v(\mathsf{hrt}, \mathsf{hrt})$ and TDM period. If $v(\mathsf{srt}, \mathsf{hrt}) < P$, where $P = N_{\mathsf{hrt}} \times SW$ (TDM period), the time duration that c_j^{srt} holds A is hidden by the time duration c_i^{hrt} holds the data and L_i^{arb}. On the other hand, if $v(\mathsf{hrt}, \mathsf{hrt}) < P$, then the SRT core will not get a *slack* slot to make a request before all $(N_{\mathsf{hrt}} - 1)$ HRT cores and c_i^{hrt} broadcast write requests. Hence, I include the term X in the first term in equation 8.22, which is 0 if $v(\mathsf{srt}, \mathsf{hrt})$ or $v(\mathsf{hrt}, \mathsf{hrt}) < P$. As a result, the interference from c_j^{srt}, $v(\mathsf{srt}, \mathsf{hrt})$, is not included in the worst-case coherence latency.

5. After the last sharer invalidates (c_i^{hrt} or c_j^{srt}), all $N_{\mathsf{hrt}} - 1$ HRT cores receive A and hold for some time duration. Each $N_{\mathsf{hrt}} - 1$ HRT core, thus incurs the latency due to data transfer at the requesting core's slot, L_{req}^{hrt}, access latency, L^{acc}, and the time duration it holds A in a valid state ($v(\mathsf{hrt}, \mathsf{hrt})$). This is given by the second term in equation 8.22.

6. The third term in equation 8.22, L_{req}^{hrt}, is added to account for the latency due to data transfer at the requesting core's slot for the core under analysis, c_i^{hrt}.

7. For the critical instance, all the remaining $(N_{\mathsf{hrt}} - 1)$ HRT cores broadcast write requests after c_j^{srt} obtained A in shared state, but before c_i^{hrt} broadcasts its write request ❸. Each core broadcasts the request in its *dedicated* slot and hence for $(N_{\mathsf{hrt}} - 1)$ HRT cores to broadcast requests, c_j^{srt} should have obtained A and started its timer countdown $(N_{\mathsf{hrt}} - 1)$ slots before c_i^{hrt}'s slot. However, the coherence latency is measured from the time when c_i^{hrt} broadcasts its write request. So, I subtract the component $(N_{\mathsf{hrt}} \times SW)$ from $v(\mathsf{srt}, \mathsf{hrt})$, as c_j^{srt} obtained A $(N_{\mathsf{hrt}} - 1)$ slots before coherence latency measurement starts. If there is no interference from SRT core, then once the timer for A in c_i^{hrt} (sharer) expires, it sends A to the next requesting HRT core, say c_k^{hrt}. Now part of L_{req}^{hrt} added for c_k^{hrt} is not included in coherence latency and hence it must be subtracted. This is given by WC bus arbitration latency, $N_{\mathsf{hrt}} \times SW$. This is given by the last term in equation 8.22.

Proof. The proof for Theorem 7 is by contradiction. It shows that the instance provided is the critical instance.

1. Suppose that the critical instance is when the HRT core under analysis c_i^{hrt}, requests for write operation on cache line A, when it is in invalid state or when it is in shared state for a time duration t, where $0 < t < v(\mathsf{hrt}, \mathsf{hrt})$. If c_i^{hrt} has A in invalid state

(A is not cached in the private cache), then $WCL_{i,arb}^{\mathsf{hrt}}$ will not include the first term $v(\mathsf{hrt},\mathsf{hrt})$ as A is not cached. If c_i^{hrt} has A in shared state, and $t > 0$, then $WCL_{i,arb}^{\mathsf{hrt}}$ will only include $v(\mathsf{hrt},\mathsf{hrt}) - t$, which is less than $v(\mathsf{hrt},\mathsf{hrt})$. Both these scenarios result in a latency less than the $WCL_{i,arb}^{\mathsf{hrt}}$ presented in the equation 8.21. So this is not the critical instance as this is not provide the worst-case arbitration latency. Hence, the critical instance must be when c_i^{hrt} requests for write on A immediately after it obtains A in shared state ($t = 0$).

2. Suppose that there is no SRT sharer after A expires in c_i^{hrt}'s cache and X is non-zero. Then the coherence latency will only include latencies incurred due to $(N_{\mathsf{hrt}} - 1)$ HRT cores. The term $v(\mathsf{srt},\mathsf{hrt})$ will not be included in coherence latency, which results in a value less than $WCL_{i,coh}^{\mathsf{hrt}}$ of equation 8.22. So this is not the critical instance that will result in the worst-case coherence latency. By contradiction, critical instance is when there is SRT sharer.

3. Suppose that the critical instance is when c_j^{srt} broadcasts a write request, instead of read on A. In such a situation, to maintain data correctness by having only a single writer at a time, c_j^{srt} must wait for c_i^{hrt}'s timer to expire. In the meanwhile, if there are other write requests from HRT cores, by fixed-priority arbitration, the requests from the other HRT cores are ordered before c_j^{srt}'s request. It will result in a scenario where c_i^{hrt}'s write request is also ordered before c_j^{srt}'s write request, resulting in c_i^{hrt} receiving A before c_j^{srt}. This results in a coherence latency for c_i^{hrt} less than $WCL_{i,coh}^{\mathsf{hrt}}$ of equation 8.22 as it will not include the interference from the SRT core. So this is not the critical instance as it does not result in worst-case coherence latency. Thus, the critical instance should be when c_j^{srt} broadcasts a read request on A and obtains it in shared state.

4. Suppose that the critical instance be when the first HRT core, say c_k^{hrt}, broadcast a read request, instead of write after c_j^{srt} obtains A in shared state. As the protocol allows for multiple sharers, c_k^{hrt} also obtains A in shared state and starts its timer, resulting in 3 sharers of A. In such a situation, the first term of equation 8.21, $v(\mathsf{hrt},\mathsf{hrt})$, overlaps with the timer values added for c_j^{srt} and c_k^{hrt}. Therefore, the resulting total latency is less than total latency obtained by adding $WCL_{i,arb}^{\mathsf{hrt}}$ and $WCL_{i,coh}^{\mathsf{hrt}}$ from equations 8.21 and 8.22. Hence, this is not the critical instance that leads to WC total latency.

5. Suppose that the critical instance is when N' HRT cores broadcast write requests to A before c_i^{hrt} broadcasts its write request after its timer expires, such that $N' <$

$(N_{\mathsf{hrt}}-1)$. The remaining $(N_{\mathsf{hrt}}-1-N')$ HRT cores either have no request or request a different cache line B. Now, c_i^{hrt} waits for N' cores to complete their request and hold A for a time duration, before it obtains A. The second term in equation 8.22 will now be $\big(N' \times (v(\mathsf{hrt},\mathsf{hrt}) + N_{\mathsf{hrt}} \times SW)\big)$. Since $N' < (N_{\mathsf{hrt}} - 1)$, this above term is less than the third term in equation 8.22, resulting in a lesser latency compared to $WCL_{i,coh}^{\mathsf{hrt}}$ of equation 8.22. So this cannot be the critical instance.

6. Supppose that the critical instance is when a core c_k broadcasts more than 1 write request on A, resulting in more than $N_{\mathsf{hrt}} - 1$ pending HRT requests before c_i^{hrt}'s write request is broadcasted. In case of an in-order core, a core cannot issue another request to a same cache line A that already has a pending request by the same core. As a result, if a core c_k has more than 1 *dedicated* slot, it can issue a read/write request to another cache line B before c_i^{hrt}'s broadcast of write request, but not A. Hence, at the maximum, there can be only $N_{\mathsf{hrt}} - 1$ pending HRT requests to the same cache line A before c_i^{hrt} broadcasts its write request. This results in the critical instance that provides the worst case coherence latency.

7. Suppose that c_j^{srt} requests for read after c_i^{hrt}'s timer expires. Since SRT cores broadcast requests only in *slack* slots, this denotes that there is a HRT core which did not make any request. This results in less than $(N_{\mathsf{hrt}}-1)$ cores requesting for write on A before c_i^{hrt} broadcasts its write request. However, as discussed previously, this is not the critical instance. The critical instance is when there are $N_{\mathsf{hrt}} - 1$ pending HRT requests to the same cache line A before c_i^{hrt} broadcasts its write request. Hence, the critical instance should be when c_j^{srt} requests for read before c_i^{hrt}'s timer expires.

8. Suppose that the SRT core, c_j^{srt}, requests for read on A few TDM slots before c_i^{hrt}'s timer expires. Now there will be an overlap between the first of equation 8.21 and first term of equation 8.22. This results in a total latency less than that obtained from the critical instance that was discussed. Hence, this is not the WC total latency and so this is not the critical instance. By contradiction, the critical instance should be when c_j^{srt} requests just when c_i^{hrt} is about to expire. There can be other SRT cores requesting for read on A before c_j^{srt}, but only the last sharer accounts for the WC coherence latency of c_i^{hrt}.

From the above discussion, it is seen that the observed arbitration latency $L_i^{arb} < WCL_{i,arb}^{\mathsf{hrt}}$ or observed coherence latency $L_i^{coh} < WCL_{i,coh}^{\mathsf{hrt}}$. Hence, I prove by contradiction that the critical instance mentioned in Theorem 7 is the instance that results in the WC total latency. □

Lemma 5. *If the timer values are assumed to be aligned to the TDM slots, then* $WCL^{\mathsf{hrt}}_{i,arb}$ *and* $WCL^{\mathsf{hrt}}_{i,coh}$ *of* c^{hrt}_i *are tighter.*

Proof. Consider that the initial values of timers $v(\mathsf{hrt},\mathsf{hrt})$ and $v(\mathsf{srt},\mathsf{hrt})$ are assigned to be multiples of TDM period and hence the timer values are aligned to TDM slots. Now, $v(\mathsf{hrt},\mathsf{hrt})$ and $v(\mathsf{srt},\mathsf{hrt})$ will be at least 1 TDM period, and so there is no need of the term X in equation 8.22. Since the timers for a cache line in a core, start their countdown after it received the cache line, the timers expire at the end of that core's slot. c^{hrt}_i's timer expires at the end of its slot, and hence it takes only $(N_{\mathsf{hrt}} - 1)$ slots to arrive at its next slot to broadcast the write request. The latency due to transfer of data at requesting core's slot, L^{hrt}_{req}, is also tighter and given by $(N_{\mathsf{hrt}} - 1) \times SW$. From the example in Figure 8.2, c^{srt}_2's timer will expire at the end of c^{hrt}_0's slot, and hence, in the worst case, L^{hrt}_{req} is only 1 SW.
Equation 8.21 can now be expressed as follows:

$$WCL^{\mathsf{hrt}}_{i,arb} = v(\mathsf{hrt},\mathsf{hrt}) + \big((N_{\mathsf{hrt}} - 1) \times SW\big) \tag{8.23}$$

Equation 8.22 can now be expressed as follows:

$$\begin{aligned} WCL^{\mathsf{hrt}}_{i,coh} = & v(\mathsf{srt},\mathsf{hrt}) \\ & + (N_{\mathsf{hrt}} - 1) \times \big(v(\mathsf{hrt},\mathsf{hrt}) + SW + ((N_{\mathsf{hrt}} - 1) \times SW)\big) \\ & + ((N_{\mathsf{hrt}} - 1) \times SW) \\ & - ((N_{\mathsf{hrt}} - 1) \times SW) \end{aligned} \tag{8.24}$$

The WC arbitration and coherence latency from equations 8.23 and 8.24 are less than the latency provided in Theorem 7 by aligning the timer values to the TDM slots. This not only makes the total WC latency tighter, but also leads to lesser hardware overhead as discussed in Chapter 7 Section 7.1.2. □

8.2.2 Bound for FRT cores

The analysis for WC latency for a FRT core is different from H-DD-NWC due to FRT requests being broadcasted in *slack* slots of HRT cores. The FRT cores broadcasts requests in the *dedicated* slots and also in the *slack* slots of HRT cores to improve performance of FRT cores. These requests broadcasted in *slack* slots will affect the WC latency bound for HRT

cores. Hence any pending FRT request to a cache line, broadcasted in the *slack* slot before a HRT request to the same cache line, must be cancelled and reissued. This is done to make sure that the HRT cores are always predictable and have a guaranteed bound. However, this affects the WC latency bound of FRT cores as it now has to account for the requests being reissued. For the special case of this arbitration, H-DD-WC-0, no slots are allocated to SRT cores and hence they do not have a guaranteed bound. Here, I derive the WCL bound for a FRT core, c_i^{frt}, when H-DD-WC is used, where N_s^{frt} is non-zero.

Bus arbitration latency: The WC bus arbitration latency of FRT cores remain the same as $WCL_{bus,arb}^{\mathsf{frt}}$ provided in Lemma 3 equation 8.11 of Section 8.1. Though H-DD-WC allows for FRT cores to utilize the *slack* slots of HRT cores, in the worst case, we can assume that there are no *slack* slots available. Then all FRT cores must broadcast their requests in *dedicated* slots of FRT cores. This results in a latency same as equation 8.11.

Coherence latency: If the access pattern of the tasks is such that the data is only read by the HRT and FRT cores, then all the cache lines exist in shared state in all the cores. As discussed in Section 8.1.2 Lemma 4, a core receives data immediately once it reaches its own slot, when the data is read-only. The use of *slack* slots only improves the average-case performance, but the worst case is when there are no *slack* slots. Hence, WC arbitration latency and WC coherence latency when H-DD-WC is used are the same as that given by H-DD-NWC in Lemma 4. This latency is not affected if data is shared or not between HRT and FRT cores, as all cores have only read accesses.

If data is not shared between HRT and FRT cores, and the FRT cores request for read and write accesses, then WC coherence latency of a FRT core depends only on the interference from other FRT cores. Hence there can be no HRT cores requesting for the same cache line as a FRT core. As a result, there are no reissue of FRT requests. Therefore the WC latencies are the same as those derived for H-DD-NWC- **Read-Write Unshared** and is given by Equation 8.17.

Theorem 8. *When FRT cores request for read and write accesses and the data is shared between HRT and FRT cores, the WC coherence latency depends on the interference from HRT and FRT cores. WC arbitration latency, $WCL_{i,arb}^{\mathsf{frt}}$, for a request issued by c_i^{frt} occurs under the critical instance when it requests for write on a cache line that c_i^{frt} has just received in the shared state(S) in its private cache. The WC coherence latency, $WCL_{i,coh}^{\mathsf{frt}}$, for a request issued by a FRT core, c_i^{frt}, occurs under the critical instance where c_i^{frt} broadcasts its request in a slack slot, after write requests from other cores, and then has to reissue its request in a dedicated slot due to a pending request from HRT core. Figure 8.8 illustrates the critical instance under which WC arbitration and WC coherence latency for a FRT core c_i^{frt} using HourGlass(H-DD-WC) are observed.*

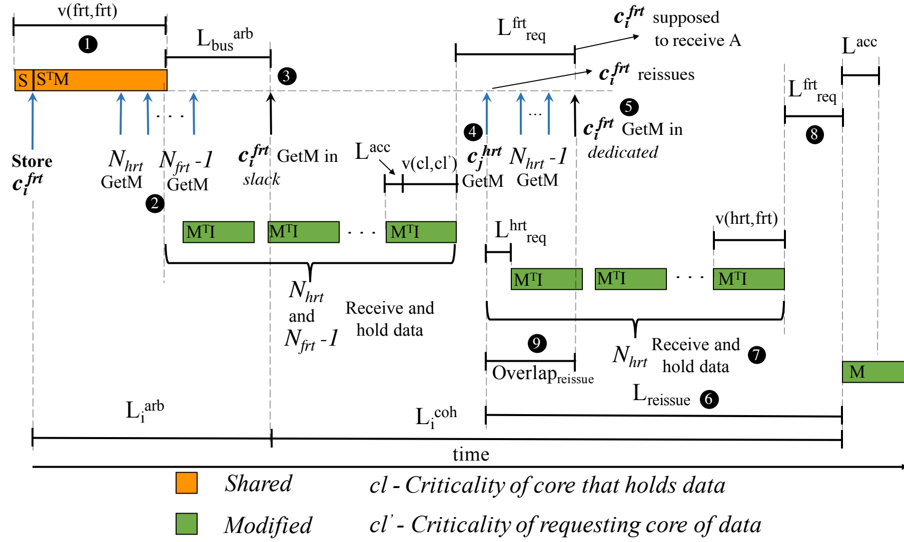


Figure 8.8: Critical Instance for H-DD-WC FRT core - Read-Write Shared.

The critical instance is explained as follows:

1. *c_i^{frt} requests for write on a cache line A which c_i^{frt} just received to be in shared state and has started its timer countdown ❶. Now c_i^{frt} has to wait for the timer for A, $v(frt, frt)$, in its own cache to expire before it can broadcast the write request on the bus. No other core holds A in a valid state.*

2. *In this duration, all HRT cores and the remaining ($N_{frt} - 1$) FRT cores broadcast write requests on A in dedicated slots ❷. Each of this core (HRT or FRT) completes its write operation and holds A for some time duration before it invalidates.*

3. *Finally c_i^{frt}'s write request is broadcasted in a slack slot of any HRT core shown as ❸ in Figure 8.8. This is different from the critical instance explained in Theorem 5 of Section 8.1, as in Section 8.1, H-DD-NWC is non-work conserving and slack slots remain idle. Since c_i^{frt}'s write request is broadcasted only after the remaining HRT and FRT write requests, c_i^{frt}'s write is serviced only after the previous pending requests are serviced.*

4. *Just before c_i^{frt}'s write request is serviced (i.e., all the previous pending requests are serviced and c_i^{frt} is about to receive A), a HRT core, c_j^{hrt}, broadcasts a write request on A ❹. Since, c_i^{frt}'s write was broadcasted in a slack slot, c_j^{hrt}'s write is prioritized*

and so c_j^{hrt} will receive A before c_i^{frt}. c_i^{frt} reissues its write request in a dedicated FRT slot ❺.

5. *Before c_i^{frt}'s write is reissued in the dedicated slot, all the remaining ($N_{hrt} - 1$) HRT cores broadcast write requests on A. So c_i^{frt}'s write is serviced only after all the N_{hrt} requests are serviced.*

The WC arbitration latency for a FRT core is given as:

***Read-Write Shared**:*

$$\begin{aligned} WCL_{i,arb}^{frt} =& v(\mathsf{frt}, \mathsf{frt}) + WCL_{bus,arb}^{frt} \\ =& v(\mathsf{frt}, \mathsf{frt}) + \left(\left\lceil \frac{N_{frt}}{N_s^{frt}} \right\rceil \times P \right) \end{aligned} \tag{8.25}$$

The WC coherence latency $WCL_{i,coh}^{frt}$ for a FRT core is given as follows:

***Read-Write Shared**:*

$$\begin{aligned} WCL_{i,coh}^{frt} =& WCL_{i,coh}^{frt}\text{-H-DD-NWC} \\ &+ L_{reissue} \end{aligned} \tag{8.26}$$

*where $WCL_{i,coh}^{frt}$-H-DD-NWC is the WC coherence latency incurred by a FRT core c_i^{frt} when HourGlass(H-DD-NWC) for **Read-Write Shared** data is used. This is discussed in Section 8.1.2 Theorem 5 and is given by the equation 8.19.*

$L_{reissue}$ is given by:

$$
\begin{aligned}
L_{reissue} =& N_{\textsf{hrt}} \times \left(L_{req}^{\textsf{hrt}} + L^{acc} + \max\{v(\textsf{hrt}, \textsf{hrt}), v(\textsf{hrt}, \textsf{frt})\}\right) \\
&+ L_{req}^{\textsf{frt}} \\
&- Overlap_{reissue} \\
\\
=& N_{\textsf{hrt}} \times \left((N_s \times SW) + SW + \max\{v(\textsf{hrt}, \textsf{hrt}), v(\textsf{hrt}, \textsf{frt})\}\right) \\
&+ (\left\lceil \frac{N_{\textsf{frt}}}{N_s^{\textsf{frt}}} \right\rceil \times P) \\
&- (N_{\textsf{hrt}} \times SW)
\end{aligned} \tag{8.27}
$$

From Equations 8.19, 8.26, 8.27, $WCL_{i,coh}^{\textsf{frt}}$ is calculated as:

$$
\begin{aligned}
WCL_{i,coh}^{\textsf{frt}} =& N_{\textsf{hrt}} \times \left((N_s \times SW) + SW + \max\{v(\textsf{hrt}, \textsf{hrt}), v(\textsf{hrt}, \textsf{frt})\}\right) \\
&+ (N_{\textsf{frt}} - 1) \times \left((\left\lceil \frac{N_{\textsf{frt}}}{N_s^{\textsf{frt}}} \right\rceil \times P) + SW + \max\{v(\textsf{frt}, \textsf{hrt}), v(\textsf{frt}, \textsf{frt})\}\right) \\
&+ N_{\textsf{hrt}} \times \left((N_s \times SW) + SW + \max\{v(\textsf{hrt}, \textsf{hrt}), v(\textsf{hrt}, \textsf{frt})\}\right) \\
&+ (\left\lceil \frac{N_{\textsf{frt}}}{N_s^{\textsf{frt}}} \right\rceil \times P) \\
&- (N_{\textsf{hrt}} \times SW)
\end{aligned}
$$

$$
\begin{aligned}
WCL_{i,coh}^{\textsf{frt}} =& 2 \times N_{\textsf{hrt}} \times \left((N_s \times SW) + SW + \max\{v(\textsf{hrt}, \textsf{hrt}), v(\textsf{hrt}, \textsf{frt})\}\right) \\
&+ (N_{\textsf{frt}} - 1) \times \left((\left\lceil \frac{N_{\textsf{frt}}}{N_s^{\textsf{frt}}} \right\rceil \times P) + SW + \max\{v(\textsf{frt}, \textsf{hrt}), v(\textsf{frt}, \textsf{frt})\}\right) \\
&+ (\left\lceil \frac{N_{\textsf{frt}}}{N_s^{\textsf{frt}}} \right\rceil \times P) \\
&- (N_{\textsf{hrt}} \times SW)
\end{aligned} \tag{8.28}
$$

The different components of the above equation are explained as follows:

1. the WC arbitration latency of a FRT core c_i^{frt} when HourGlass(H-DD-WC) is used, $WCL_{i,arb}^{\mathsf{frt}}$, is the same as the WC arbitration latency when HourGlass(H-DD-NWC) is used.

2. The WC coherence latency of a FRT core c_i^{frt} when HourGlass(H-DD-WC) is used, $WCL_{i,coh}^{\mathsf{frt}}$-H-DD-WC, includes the latency incurred due to reissue of c_i^{frt}'s write request and the WC coherence latency of FRT core when HourGlass(H-DD-NWC) is used, $WCL_{i,coh}^{\mathsf{frt}}$-H-DD-NWC. This is given by equation 8.26. $WCL_{i,coh}^{\mathsf{frt}}$-H-DD-NWC is included because the critical instance explained here is similar to the critical instance explained in Theorem 5 of Section 8.1, with the only difference being that c_i^{frt} broadcasts the write request in a *slack* slot. However, since it is broadcasted in a *slack* slot, when there is a HRT request, c_i^{frt} reissues and this leads to additional latency $L_{reissue}$ ❻.

3. The derivation of $WCL_{i,coh}^{\mathsf{frt}}$-H-DD-NWC is explained in Theorem 5 of Section 8.1.

4. $L_{reissue}$ is due to the issue of a write request from a HRT core, c_j^{hrt}. In the worst case, the first HRT core in the TDM schedule issues a write request on A, just before c_i^{frt} is about to receive A. As a result, c_i^{frt} reissues its write request. However, this write request is reissued in the *dedicated* slot of FRT cores. In the meanwhile, all the remaining $(N_{\mathsf{hrt}} - 1)$ HRT cores, that are scheduled before the *dedicated* slots for FRT cores, broadcast write requests on A. So c_i^{frt}'s write is serviced only after all N_{hrt} HRT cores are serviced.

5. The first term of $L_{reissue}$ in equation 8.27 accounts for interference due to all N_{hrt} HRT cores, that complete their write operation and hold the data for some time duration ❼. Each HRT core also includes L_{req}^{hrt} latency, as explained in Theorem 2 of Section 8.1.1.

6. After all N_{hrt} HRT cores are serviced, c_i^{frt} is serviced in its own slot. This latency to receive data in its own slot is given by L_{req}^{frt} ❽ (second term in the equation for $L_{reissue}$).

7. For the critical instance, all N_{hrt} HRT cores broadcast write requests before c_i^{frt} is supposed to receive A. However, the first HRT request will cause a reissue and that latency is given by $L_{reissue}$. Thus there is an overlap ($\mathrm{Overlap_{reissue}}$) of L_{reissue} with

L_{req}^{frt} for c_i^{frt} before it was reissued (i.e., L_{req}^{frt} from equation 8.19). This Overlap$_{\text{reissue}}$ is shown as ❾ in Figure 8.8 and is given by $(N_{\mathsf{hrt}} \times SW)$.

Proof. The proof for Theorem 8 is by contradiction. It shows that the critical instance provided is the critical instance.

1. The initial part of the critical instance is similar to the critical instance considered in Section 8.1 Theorem 5 which is already proved. Hence, I prove only the reissue part of the critical instance.

2. Suppose that c_i^{frt} broadcasted its request in a *dedicated* slot. Now, even if a HRT core broadcasts a write request before c_i^{frt} is serviced, the requests are not reordered as c_i^{frt} broadcasted its request in a *dedicated* slot. So the reissue component, L_{reissue} is not added. This results in a latency less than $WCL_{i,coh}^{\mathsf{frt}}$ in equation 8.27. So this is not the critical instance. For critical instance, c_i^{frt} should broadcast its request in a *slack* slot.

3. Suppose that there is another FRT core requesting for write on A, instead of a HRT core. In this case, the FRT core will not be reissued as the FRT cores are serviced in first-come first-serve manner. So this will not include L_{reissue}, resulting in a latency less than $WCL_{i,coh}^{\mathsf{frt}}$ in equation 8.27. Thus the critical instance should have HRT cores requesting for A before c_i^{frt} receives A.

4. Suppose that only N' HRT cores broadcasted write requests on A, before c_i^{frt} reissues its request in a *dedicated* slot, where $N' < N_{\mathsf{hrt}}$. So only N' write requests are serviced before c_i^{frt} is serviced. Since $N' < N_{\mathsf{hrt}}$, the latency incurred due to N' cores is less than the latency due to N_{hrt}, in the equation for L_{reissue}. This results in a coherence latency less than $WCL_{i,coh}^{\mathsf{frt}}$ in equation 8.27. This is not the critical instance. Critical instance is when all N_{hrt} broadcast write requests on A before c_i^{frt} reissues its request in a *dedicated* slot.

The observed coherence latency, explained above, is less than $WCL_{i,coh}^{\mathsf{frt}}$ in equation 8.27. Thus by contradiction, I prove that the critical instance considered in Theorem 8 is the instance that results in WC coherence latency. □

Theorem 9. *The total worst-case latency for a request issued by a* FRT *core* c_i^{frt} *is given by the WC arbitration latency, WC coherence latency and the WC access latency for* c_i^{frt}.

From the discussion on arbitration latency and Theorem 8, the WCL_i^{frt} *per request for* c_i^{frt} *is given as follows:*

$$WCL_i^{\mathsf{frt}} = WCL_{i,arb}^{\mathsf{frt}} + WCL_{i,coh}^{\mathsf{frt}} + L^{acc} \tag{8.29}$$

Chapter 9

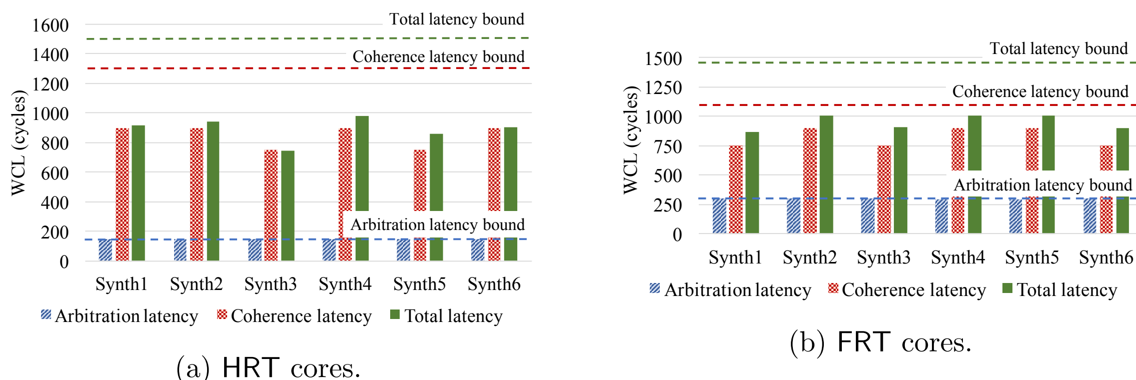
Evaluation

HourGlass is implemented in gem5 [7], which is a cycle accurate micro-architectural simulator. I use Ruby memory model to implement the cache, memory subsystem, and coherence protocol with high precision. Ruby Memory model of gem5 has flexibility to implement different cache coherence protocols and accurately models the different states and state transitions. I consider a multi-core setup consisting of four x86 in-order cores that run at 2GHz. I use two HRT cores, and two FRT or SRT cores. Every core has a separate 16kB direct mapped private L1-data (L1-D) and instruction (L1-I) cache with a cache line size of 64B. Access to the private L1 caches takes 3 cycle latency. All the cores share a 8-way 1MB set-associative last-level cache (LLC). Accesses to the LLC are assumed to be perfect and incur an access latency of 50 cycles. This allows the evaluation of this work to focus on measuring metrics relevant to cache coherence. The TDM slot width in the TDM schedule is set to 50 cycles. I assume that main-memory access overheads can be calculated using prior approaches such as [20], and are additive to the coherence latencies derived in this work [33]. The bus manages accesses between cores and the shared memory using one of the arbitration schemes discussed in Chapter 5. This is represented as HourGlass(`arb`), where `arb` is the arbitration scheme chosen. For evaluation, I use SPLASH-2 [31], a multi-threaded benchmark suite, and multi-threaded synthetic benchmarks for maximum sharing of data across cores. I also use a micro-benchmark suite, Synchrobench [17], and a multi-core performance suite for automotive processors, AutoBench 2.0 [1]. I run these benchmarks using four threads, and map each thread to a core.

9.1 Data correctness and protocol verification

I verify the correctness of HourGlass using synthetic benchmarks, Synchrobench, AutoBench 2.0 and SPLASH-2 benchmarks. For synthetic benchmarks, I run the same set of instructions across all cores on the same data simultaneously. This results in maximum sharing of data and stresses all the states and state transitions of HourGlass. This also causes intereference among requests from different criticality levels. I manually verified that all state transitions and timer events of HourGlass are correctly exercised. I verified all the corner cases when using different arbitration schemes on all the benchmark suites. I observe that the WCL per request for HRT cores and FRT cores are within the analytical bound, indicating that no unpredictable scenario occurred. This verifies the correctness of the protocol. However, a cache coherence protocol must also ensure data correctness by keeping data coherent across the cores. For data correctness of HourGlass, I check the output of SPLASH-2 benchmarks. These benchmarks have in-built single threaded verification routines that check the output generated from the multi-threaded workload.

9.2 Bounding memory access latencies

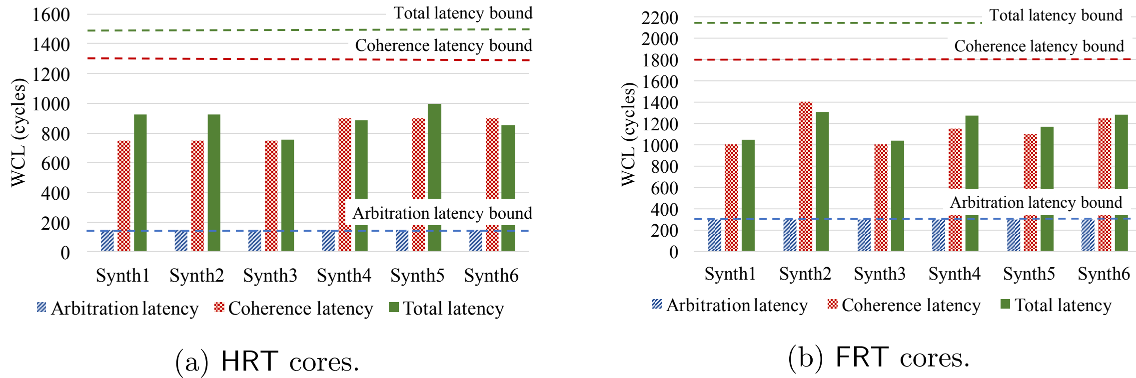


(a) HRT cores.

(b) FRT cores.

Figure 9.1: Observed WCL components for HourGlass(H-DD-NWC).

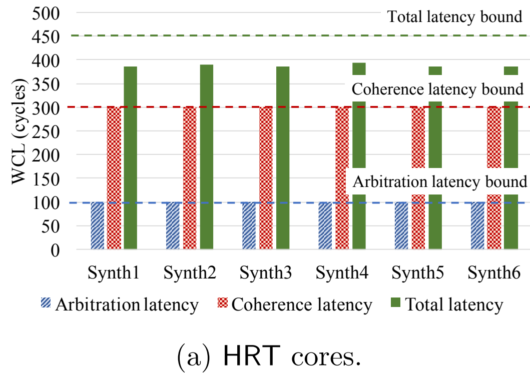
I show that HourGlass, with all the different arbitration schemes, have a bound on the access latency to shared data. Figures 9.1 and 9.2 show the observed worst-case request access latency, and the individual latency components such as the coherence latency, and arbitration latency for HRT and FRT cores observed when HourGlass(H-DD-NWC) and HourGlass(H-DD-WC) are used. For HourGlass(H-DD-WC-0), I show the observed latency



(a) HRT cores.

(b) FRT cores.

Figure 9.2: Observed WCL components for HourGlass(H-DD-WC).



(a) HRT cores.

Figure 9.3: Observed WCL components for HourGlass(H-DD-WC-0).

components only for HRT cores, as SRT cores do not have a bound in Figure 9.3. In this evaluation, I configure the initial timer values for all timer configurations as 1 TDM period. The TDM period P depends on the arbitration scheme chosen. I use synthetic benchmarks that perform the same set of instructions on all cores. For all the benchmarks: SPLASH-2, Synchrobench and AutoBench 2.0, I observe that the different latencies per request are bounded and well within the analytical bound. This is because they do not stress the worst case instances due to low sharing of data across the cores. Hence, I show the observed WCL for synthetic workloads that stress the coherence protocol, and the WCL and their respective components for all benchmarks are within the analytical bounds.

Configuration	Timer values	Description
HourGlass(ALL-DD) (1)	$v(\mathsf{hrt},\mathsf{hrt}) = 1 * P$, $v(\mathsf{hrt},\mathsf{srt}) = 1 * P$, $v(\mathsf{srt},\mathsf{hrt}) = 1 * P$, $v(\mathsf{srt},\mathsf{srt}) = 1 * P$	Each HRT, FRT or SRT core is allocated a *dedicated* slot
HourGlass(H-DD-NWC) (2,4,2,1)	$v(\mathsf{hrt},\mathsf{hrt}) = 2 * P$, $v(\mathsf{hrt},\mathsf{frt}) = 4 * P$, $v(\mathsf{frt},\mathsf{hrt}) = 1 * P$, $v(\mathsf{frt},\mathsf{frt}) = 2 * P$	Each HRT core is allocated a *dedicated* slot and $N_s^{\mathsf{frt}} = 1$ slot is allocated for FRT cores; Non-work-conserving
HourGlass(H-DD-WC) (2,4,2,1)	$v(\mathsf{hrt},\mathsf{hrt}) = 2 * P$, $v(\mathsf{hrt},\mathsf{frt}) = 4 * P$, $v(\mathsf{frt},\mathsf{hrt}) = 1 * P$, $v(\mathsf{frt},\mathsf{frt}) = 2 * P$	Each HRT core is allocated a *dedicated* slot and $N_s^{\mathsf{frt}} = 1$ slot is allocated for FRT cores; Work-conserving
HourGlass(H-DD-WC-0) (2,4,2,1)	$v(\mathsf{hrt},\mathsf{hrt}) = 2 * P$, $v(\mathsf{hrt},\mathsf{srt}) = 4 * P$, $v(\mathsf{srt},\mathsf{hrt}) = 1 * P$, $v(\mathsf{srt},\mathsf{srt}) = 2 * P$	Each HRT core is allocated a *dedicated* slot and no slots are allocated for SRT cores; Work-conserving

Table A1: HourGlass configurations.

9.3 Comparison of per request WCL bounds of HourGlass with other approaches

Figure 9.4 shows the observed WCL for a request from a HRT core deployed using real-time approaches, PMSI and HourGlass. The other approaches used in real-time systems for predictable data sharing include *uncache-all* that does not cache any data in the private caches, *uncache-shared* [10] that caches only private data and accesses shared data from shared memory (main memory), and *task mapping* that maps all tasks that share data to the same core [10]. By using different arbitration schemes, different versions of HourGlass are obtained. I tabulate the different HourGlass versions along with their timer configurations used in this evaluation in Table A1. The initial timer values are set based on the TDM period of the arbitration scheme P.

The conventional PMSI uses TDM arbitration, by allocating *dedicated* slots for all cores (HRT, FRT or SRT) and does not support cache-to-cache transfer. Since HourGlass has cache-to-cache transfer, I modify PMSI to support cache-to-cache transfers and compute the analytical WCL for a memory request under this variant of PMSI. I also compare the

different versions of HourGlass with a configuration that sets the initial timer values to zero (HourGlass (0)). By setting the timer values to zero, I assume a version of HourGlass that does not use timers and hence does not hold cache lines in the private cache for certain time duration. For this evaluation, I use one of the synthetic benchmarks to observe the WCL for a request from a HRT core.

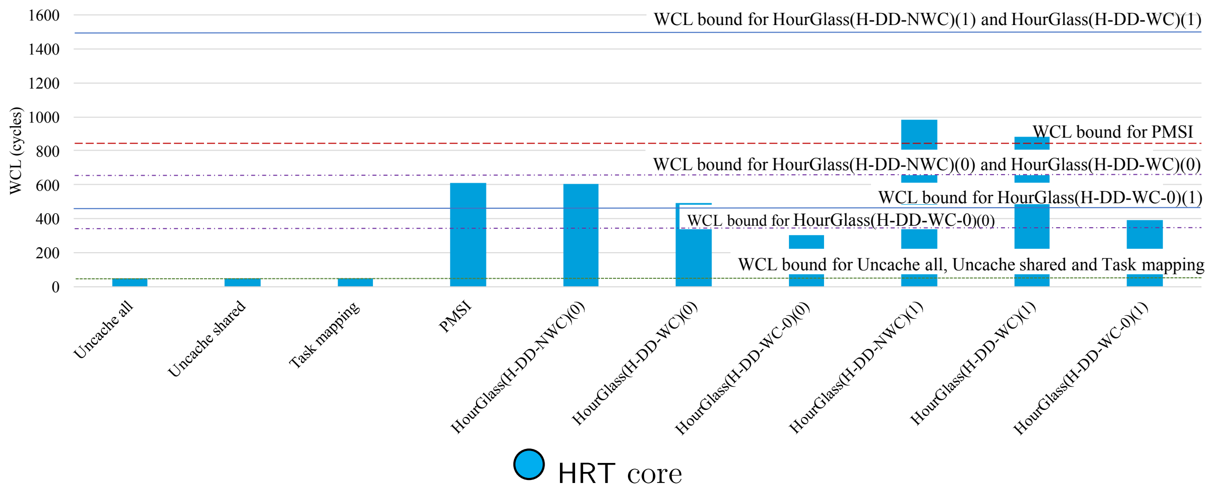


Figure 9.4: Observed per request WCL of HRT cores for all real-time approaches.

In *uncache-all*, *uncache-shared* and *task mapping* approaches, the worst-case latency for a request is observed to be the latency to obtain data from main memory. However, this results in degraded overall performance as most requests are serviced from main memory. PMSI and HourGlass, on the other hand, provide significant performance benefits, by allowing simultaneous access of shared data from private caches. This is discussed in Section 9.4. PMSI allocates a *dedicated* slot for all HRT and FRT or SRT cores and so does not differentiate between different criticality levels. WCL for HRT cores, thereby, incurs additional arbitration latency for arbitrating across FRT or SRT cores. When timer values are zero, HourGlass, with any one of the criticality-aware arbitration schemes, provides tighter bounds on the WCL for HRT cores compared to that of PMSI. Few slots are allocated for FRT cores in case of H-DD-NWC and H-DD-WC and hence it results in lower arbitration latencies for HRT cores and thereby lower total WCL per request. In H-DD-WC-0, no slots are allocated for SRT cores and hence it does not have a bound. But this also provides tighter WCL compared to PMSI. FRT or SRT cores benefit from improved average-case performance and not tighter WCL bounds. The timers in HourGlass are added for this purpose, i.e., to improve the performance of FRT or SRT cores. In Figure 9.4, HourGlass (1) denotes HourGlass with timers. It is observed that HourGlass with timers increases the WCL

bound for HRT cores. However, they provide improvement in performance of FRT or SRT cores. I discuss the trade-offs between WCL bound for HRT cores and performance of FRT or SRT cores in Section 9.5. However, not all versions of HourGlass with timers increase the WCL bound. HourGlass(H-DD-WC-0) (1) provides tighter WCL bound for HRT cores than PMSI. This provides both tighter WCL for HRT cores and improved average-case performance for SRT cores. Thus, compared to PMSI, HourGlass is a criticality-aware cache coherence protocol that satisfies the requirements of HRT and FRT or SRT cores.

9.4 Comparison with other approaches

I compare the performance of HourGlass with conventional cache coherence protocols such as the MSI and MESI protocols, state-of-the-art real-time cache coherence protocol PMSI, and other real-time approaches for sharing data predictably, such as *uncache-all*, *uncache-shared* and *task mapping* in Figure 9.5. The different HourGlass versions along with their timer configurations used in this evaluation are tabulated in Table A1. Figure 9.5 compares the slowdown experienced by different approaches to MESI cache coherence protocol for SPLASH-2 benchmarks. *uncache-all* has the largest slowdown compared to the rest of the

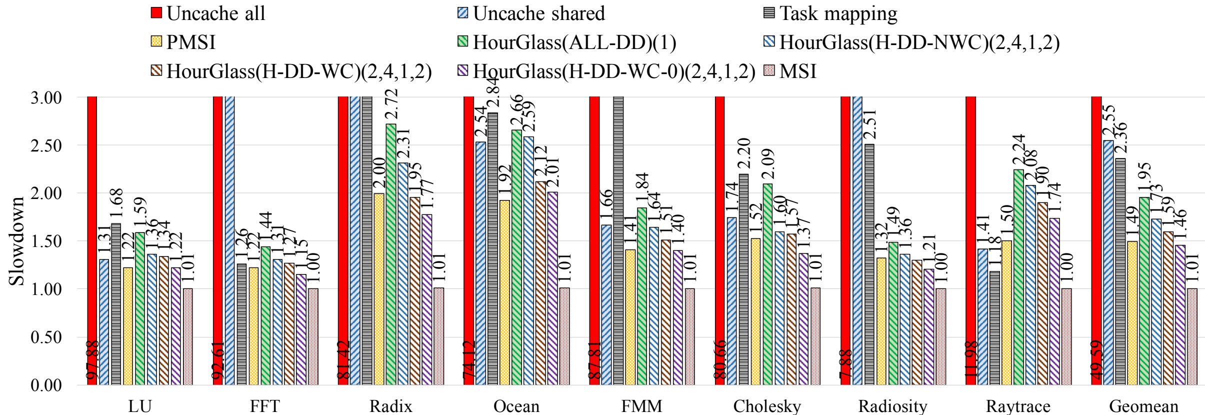


Figure 9.5: Total execution time slowdown compared to MESI protocol.

approaches and protocols. On average it suffers from a 49.59× slowdown as the SPLASH-2 benchmarks exhibit data reuse. This is because every data access incurs the long memory access latency. *uncache-shared* performs significantly better than *uncache-all* as some of the private data reuse are cache hits in the private caches. On average, *uncache-shared* suffers a slowdown of 2.55× compared to the MESI protocol. *Task mapping* has all tasks

that share data mapped to the same core. So if all tasks share data, then it is similar to running the application on a single core, and hence it does not observe the performance benefits from multi-core platform. It suffers 2.36× slowdown compared to MESI protocol. PMSI protocol outperforms previous real-time approaches as it allows for multiple copies of shared and private data to be cached simultaneously in the private caches of different cores. Hence, PMSI experiences less average slowdown (49%) compared to the previous approaches.

All versions of HourGlass perform much better than *uncache-all* and *uncache-shared* on average. However, in benchmarks such as LU and *raytrace*, it is seen that *uncache-shared* performs better than HourGlass and even PMSI. In these benchmarks, reuse of shared data is minimal as *uncache-shared* approach has a slowdown of only about 1.35× compared to MESI. Hence, acccessing the shared data from main memory is better than using a cache (that causes invalidation and write back to memory when requested by another core), resulting in better performance of *uncache-shared*.

When compared to PMSI, HourGlass(ALL-DD) incurs on average a 30% increase in execution time compared to PMSI. HourGlass(ALL-DD) considers all cores to be of the same criticality level and dedicates a slot for each core. It holds the cache lines in private caches for a time duration (1 TDM period in this evaluation). Although SPLASH-2 benchmarks exhibit data reuse and locality, the data reuse do not occur within the timer duration. Hence, cores hold on to cache lines for a duration where they may not be reused, and unnecessarily stall cores requesting for the same cache line. SPLASH-2 benchmarks include synchronization of threads in the form of barriers that result in HRT cores waiting for FRT or SRT cores to satisfy the synchronization condition. Hence, the timers, with calls to synchronization routines in SPLASH-2 benchmarks, results in HourGlass(ALL-DD)'s performance to be worse than PMSI. For similar reasons, HourGlass(H-DD-NWC) also performs worse than PMSI with 16% increase in execution time compared to PMSI. Though HourGlass(H-DD-NWC) is criticality-aware, due to synchronization, the HRT cores are dependent on the completion of requests from FRT or SRT cores and hence with timers, it incurs a slowdown of 1.73× compared to MESI protocol.

HourGlass(H-DD-WC) uses work-conserving arbitration, allowing FRT cores to utilize the *slack* slots. As a result, HourGlass(H-DD-WC) performs better than HourGlass(H-DD-NWC) with 8% decrease in execution time. Almost all SPLASH-2 benchmarks partition the data across cores such that there is minimal sharing, and have high data reuse resulting in high cache hit rates. Hence, there are abundant slack slots for FRT cores to satisfy their memory requests, thereby improving performance when work-conserving arbitration is used. For SPLASH-2 benchmarks, it is observed that the maximum utilization of a HRT slot by a HRT core to broadcast its requests is only about 30%, indicating that it results in a large

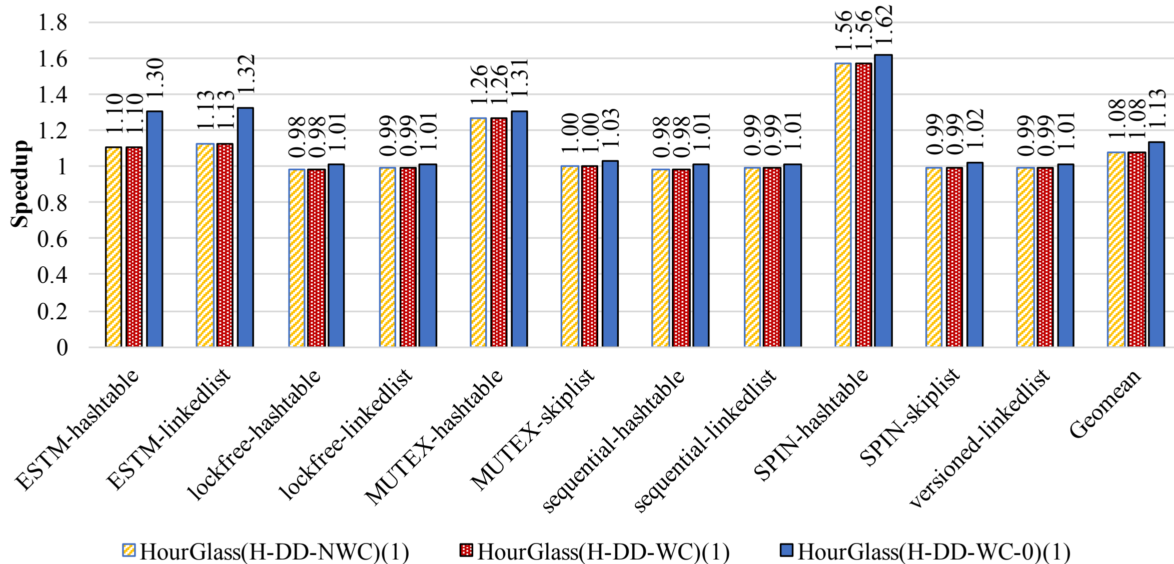


Figure 9.6: Speedup compared to PMSI protocol - Synchrobench.

number of *slack* slots.

SRT cores are not required to have a guaranteed bound, but only benefit from performance. In such an application that has SRT cores, HourGlass(H-DD-WC-0) is used. HourGlass(H-DD-WC-0) does not allocate slots for SRT cores. From the evaluation, it is seen that HourGlass(H-DD-WC-0) performs slightly better than PMSI. It decreases execution time by 2% on average over PMSI. Due to synchronization of threads, HourGlass(H-DD-WC-0) results in comparable execution times to PMSI for the SPLASH-2 benchmark suite. If the calls to synchronization routines are eliminated, then the HRT cores could complete earlier in the HourGlass than in PMSI.

For Synchrobench and AutoBench 2.0 benchmarks, all versions of HourGlass perform better than PMSI. Figure 9.6 compares the performance of HourGlass with PMSI for Synchrobench benchmark. I set the intial timer values for all configurations as 1 TDM period, denoted as HourGlass(arb) (1), where arb is one of the criticality-aware arbitration schemes: H-DD-NWC, H-DD-WC or H-DD-WC-0. All versions of HourGlass perform either comparable to PMSI or better than PMSI. I show only few instances of Synchrobench in Figure 9.6. On average, HourGlass(H-DD-NWC) and HourGlass(H-DD-WC) have 1.08× speedup over PMSI. HourGlass(H-DD-WC-0) performs 13% better than PMSI. This shows that in Synchrobench, the FRT and SRT cores benefit from the use of timers and also reduced arbitration latency due to reduced number of slots allocated in the TDM schedule.

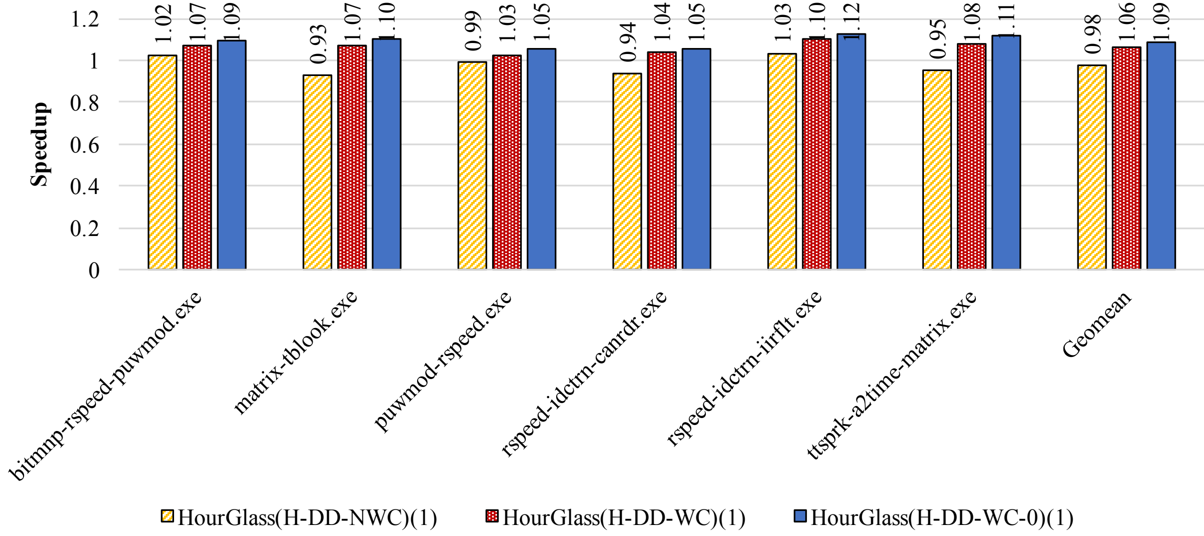


Figure 9.7: Speedup compared to PMSI protocol - AutoBench 2.0.

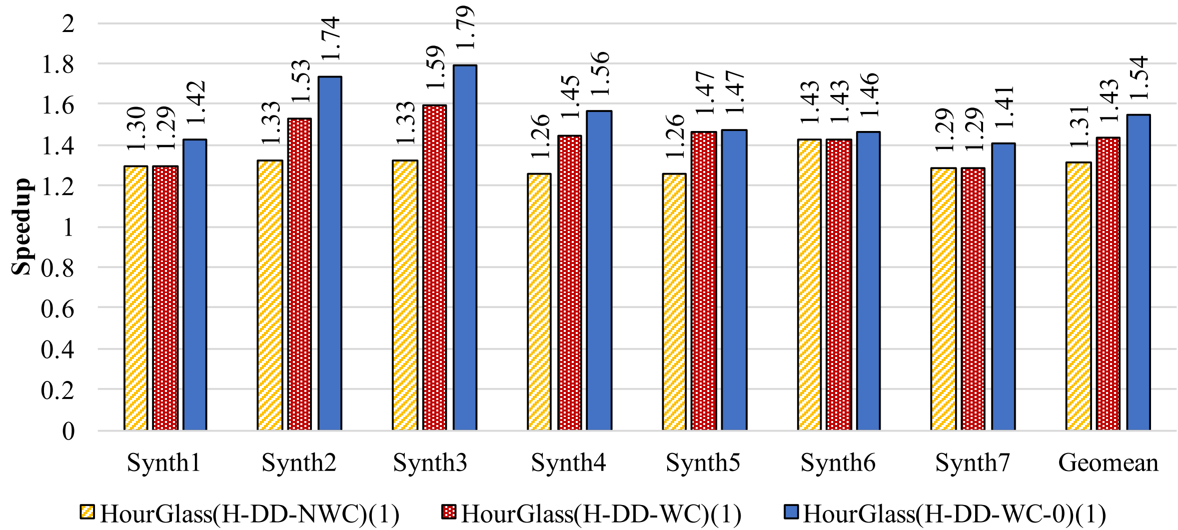


Figure 9.8: Speedup compared to PMSI protocol - Synthetic benchmarks.

In Figure 9.7, I compare the performance of HourGlass with PMSI when AutoBench 2.0 benchmark is used. For AutoBench 2.0, HourGlass(H-DD-NWC) performs worse than PMSI by 2% increase in execution time on average. However, HourGlass(H-DD-WC) and HourGlass(H-DD-WC-0) perform 6% and 9% better than PMSI. This shows that they benefit from the use of *slack* slots.

I also use synthetic benchmarks to stress HourGlass and observe its benefits. Figure 9.8 provides a comparison of HourGlass with PMSI for synthetic benchmarks with initial timer values for HourGlass set as 1 TDM period. With Synthetic benchmarks that have high locality of requests from HRT and FRT or SRT cores, it is observed that the timers improve performance of HourGlass compared to PMSI. HourGlass(H-DD-NWC) performs 1.31× better than PMSI. HourGlass(H-DD-WC), due to the use of *slack* slots, perform even better with 1.43× speedup compared to PMSI. HourGlass(H-DD-WC-0) has a speedup of 1.54× compared to PMSI. Since no slots are allocated for SRT cores in HourGlass(H-DD-WC-0), it results in tighter arbitration latency and hence further decrease in execution time.

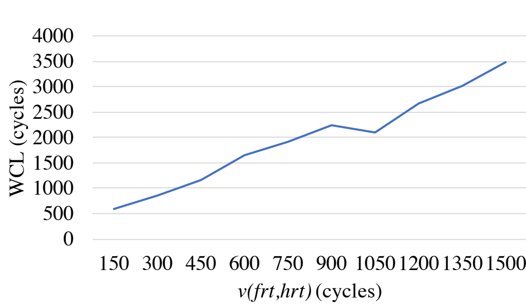
9.5 Effect of timers on the performance of {FRT,SRT} cores

Recall that timing guarantees are necessary only for HRT cores. FRT and SRT cores, on the other hand, do not require timing guarantees but it is desirable to improve their performance. This is because applications running on FRT or SRT cores may have high locality and hence high execution throughput. With HourGlass, improved performance can be achieved for FRT or SRT cores at the expense of increasing the WCL bound for HRT cores.

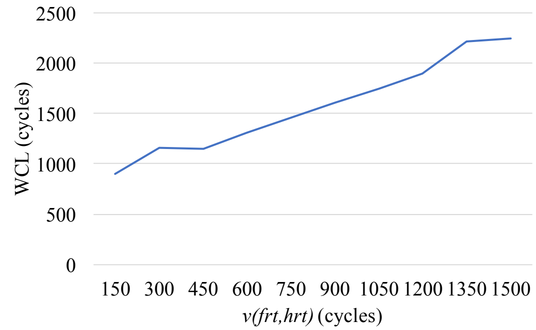
The performance of FRT or SRT cores can be improved by increasing the time duration these cores hold data. However, this results in an increase in the WCL bound for HRT cores. One major contribution of HourGlass is the support for varying the timer configurations to provide trade-offs between performance of FRT and SRT cores and WCL bound for HRT cores. Increasing the timer values for HRT cores, $v(\mathsf{hrt}, \mathsf{hrt})$ and $v(\mathsf{hrt}, \mathsf{frt})$ or $v(\mathsf{hrt}, \mathsf{srt})$, increases the WCL bound for HRT cores. However, this decreases the interference from FRT or SRT cores on HRT cores, as HRT cores have higher priority. This results in degraded performance for FRT or SRT cores as it provides more opportunities for HRT cores to prioritize requests from FRT or SRT cores broadcasted in *slack* slots.

For performance improvement of FRT or SRT cores, the timers for these cores, $v(cl_2, \mathsf{hrt})$ and $v(cl_2, cl_2)$ where cl_2 is frt or srt, should be varied. Varying $v(cl_2, cl_2)$ affects the number

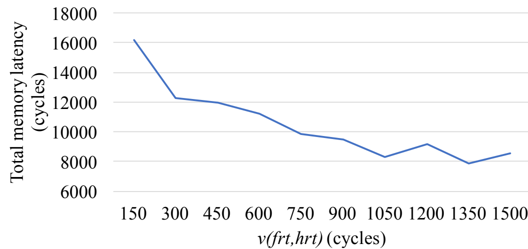
of cache hits and hence performance, when there are requests only from cl_2 cores. However, with requests from HRT cores, if $v(cl_2, \mathsf{hrt})$ is low, then the timer in cl_2 core expires earlier and thus it has to wait for its next slot for further accesses to the same data. In this evaluation, I focus on varying the initial value for $v(cl_2, \mathsf{hrt})$ and observe its effect on the performance of cl_2 cores and WCL bound of HRT cores. Higher values of $v(cl_2, \mathsf{hrt})$ increases the WCL of HRT cores. This is because a request to shared data that is present in the private cache of cl_2 cores has to wait for a maximum of $v(cl_2, \mathsf{hrt})$ before completing its request. However, increasing $v(cl_2, \mathsf{hrt})$ allows cl_2 cores to hold the data longer and thus increases the number of cache hits on shared data in their private caches. This leads to performance improvement. I observe the effect on performance and WCL by varying $v(cl_2, \mathsf{hrt})$ for all versions of HourGlass. I use a synthetic benchmark that shows an improvement in performance of FRT or SRT cores.



(a) WCL of HRT cores.



(b) WCL of FRT cores.

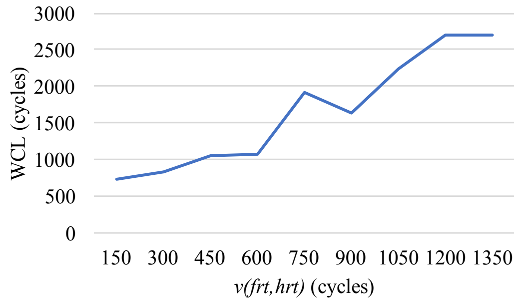


(c) Total Memory access latency of FRT cores.

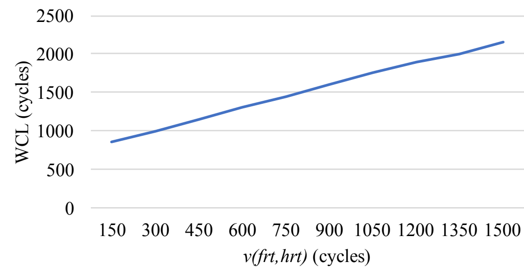
Figure 9.9: Effect of timers in HourGlass(H-DD-NWC).

9.5.1 Effect on HourGlass(H-DD-NWC)

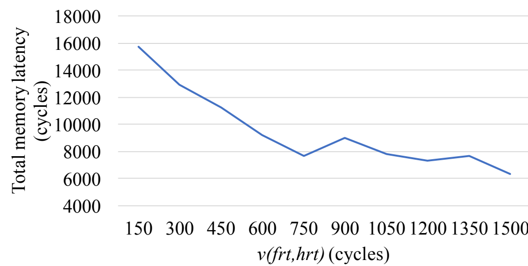
Figure 9.9 shows the effect of varying $v(\mathsf{frt}, \mathsf{hrt})$ on WCL bound of HRT and FRT cores and total memory latency of requests from FRT cores. From Chapter 8, we know that the WCL bounds for HRT and FRT cores depend on $v(\mathsf{frt}, \mathsf{hrt})$. $v(\mathsf{frt}, \mathsf{hrt})$ is varied in terms of TDM period, which is $N_s \times SW = 150$ for HourGlass(H-DD-NWC). The other timer configurations are kept constant. Figures 9.9a and 9.9b show an increase in observed WCL for HRT and FRT cores with increasing $v(\mathsf{frt}, \mathsf{hrt})$. Though the observed WCL for FRT cores is increased, the total memory access latency for FRT cores is decreased. This total memory latency is given by the latency incurred by all memory requests from FRT cores. Increasing the timer duration, increases the number of cache hits and thus the total latency is decreased. As a result, the average-case performance of FRT cores is improved. Depending on the WCL requirements of HRT cores of the application, one can set the timer values to obtain improved performance for FRT cores.



(a) WCL of HRT cores.

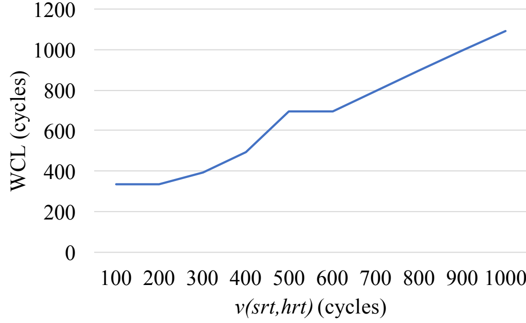


(b) WCL of FRT cores.

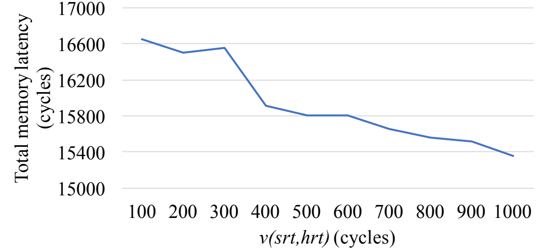


(c) Total memory access latency of FRT cores.

Figure 9.10: Effect of timers in HourGlass(H-DD-WC).



(a) WCL of HRT cores.



(b) Total memory access latency of SRT cores.

Figure 9.11: Effect of timers in HourGlass(H-DD-WC-0).

9.5.2 Effect on HourGlass(H-DD-WC)

Figure 9.10 shows the effect of varying $v(\mathsf{frt}, \mathsf{hrt})$ on WCL bound of HRT and FRT cores and total memory latency of FRT cores using HourGlass(H-DD-WC). Here, N_s^{frt} is non-zero for the arbitration and hence FRT cores have a WCL bound. From Figures 9.10a, 9.10b and 9.10c, we see that increasing $v(\mathsf{frt}, \mathsf{hrt})$, increases the number of cache hits for FRT cores, and hence the total memory latency decreases. However, this also increases the observed WCL for HRT and FRT cores. For the same benchmark and same variation in timer values, HourGlass(H-DD-WC) has improved performance compared to HourGlass(H-DD-NWC) as total memory latency is further decreased in case of HourGlass(H-DD-WC). This is because, HourGlass(H-DD-WC) also benefits from the use of *slack* slots.

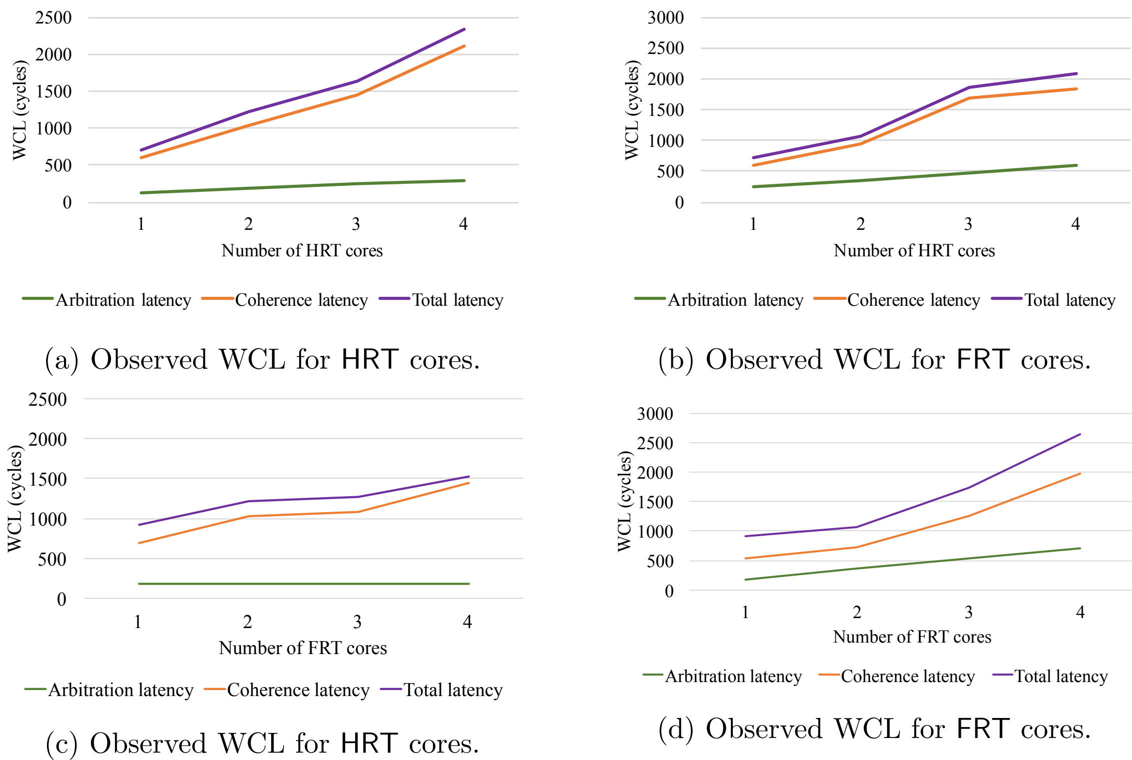
9.5.3 Effect on HourGlass(H-DD-WC-0)

In H-DD-WC-0 arbitration, $N_s^{\mathsf{frt}} = 0$ and hence SRT cores do not have a guaranteed bound. Figure 9.11 shows the effect of varying $v(\mathsf{srt}, \mathsf{hrt})$ on WCL bound of HRT and SRT cores and the memory access latency of SRT cores using HourGlass(H-DD-WC-0). Figures 9.11a and 9.11b show the effect on observed WCL of HRT cores and total memory latency of SRT cores with increasing $v(\mathsf{srt}, \mathsf{hrt})$. Thus, the timers in HourGlass provide both bounds on WCL for HRT cores, and performance benefits for SRT cores at the expense of increasing the WCL of HRT cores.

9.6 Scalability

In this section, I analyze the effects of varying the number of HRT and FRT or SRT cores on the WCL of HRT and FRT cores, when HourGlass is used. The dependence of WCL bounds on the number of cores vary for different criticality-aware arbitration schemes. I show the effect on WCL bounds by varying the number of HRT cores and FRT or SRT cores for each protocol version. For this evaluation, I use one of the synthetic benchmarks to distinctly highlight the trends as the synthetic benchmarks have more sharing of data across the cores.

9.6.1 Effect on HourGlass(H-DD-NWC)



(a) Observed WCL for HRT cores.

(b) Observed WCL for FRT cores.

(c) Observed WCL for HRT cores.

(d) Observed WCL for FRT cores.

Figure 9.12: Effect of scalability in HourGlass(H-DD-NWC).

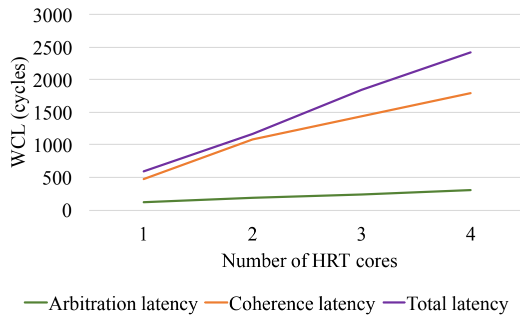
In HourGlass(H-DD-NWC), the WCL bound of HRT cores depend on the number of HRT cores and also on the number of FRT cores depending on the timer duration. Figure 9.12a

and 9.12b show the variation in WCL bounds of HRT and FRT cores respectively by increasing the number of HRT cores and keeping the number of FRT cores constant. The number of *dedicated* slots allocated for FRT cores is also kept constant. In Figure 9.12a, the WCL bound for HRT cores increase with increase in number of HRT cores. The arbitration latency is proportional to the TDM period, and since each HRT core is allocated a *dedicated* slot, increase in number of HRT cores increases the TDM period. The coherence latency increases quadratically with increase in HRT cores, as there is an increased number of HRT cores that cause interference. The total WCL latency is dominated by coherence latency. The WCL bounds for FRT cores also increase with increase in number of HRT cores. The number of HRT cores affect the TDM period, which in turn affects the arbitration latency of FRT cores. The coherence latency of FRT cores also depend on the number of HRT cores interfering with the requests from FRT cores. This is shown in Figure 9.12b.

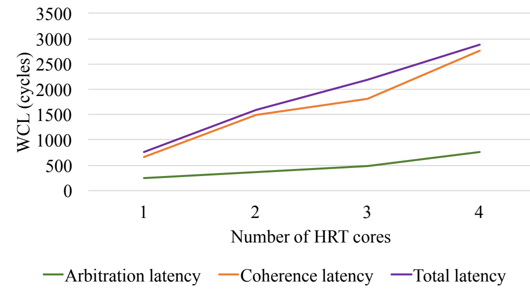
The effect of varying the number of FRT cores has a different impact on the WCL bounds of HRT and FRT cores. Since the number of *dedicated* slots for FRT cores remain the same, the TDM period remains constant with increase in the number of FRT cores. Hence arbitration latency of HRT cores remains constant with increase in number of FRT cores. Coherence latency, on the other hand, depends on the maximum number of FRT cores that can interfere with requests from HRT cores (N'_{frt} as discussed in Chapter 8). In the scenario where one slot is allocated to FRT cores and the timer duration of HRT cores is set to 1 TDM period, N'_{frt} can be 2 at the maximum. Hence in Figure 9.12c, there is an increase in coherence latency, and thus total latency, of HRT cores when the number of FRT cores increase from 1 to 2. Afterwards, as the number of FRT cores increase more than N'_{frt}, it has no effect on the WCL bounds of HRT cores. Figure 9.12d shows the WCL trends of FRT cores with increase in the number of FRT cores. Since the number of *dedicated* slots for FRT cores is kept constant, increase in number of FRT cores results in increase in the arbitration latency for FRT cores. The coherence latency of FRT cores also depends on the number of FRT cores and hence it varies quadratically with increase in number of FRT cores. This also increases the total latency of FRT cores quadratically.

9.6.2 Effect on HourGlass(H-DD-WC)

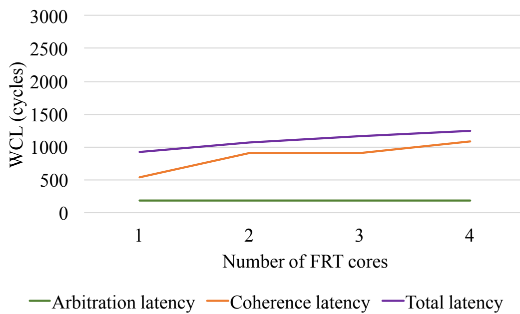
Figure 9.13 shows the variation in WCL of HRT and FRT cores with variation in the number of HRT and FRT cores, when HourGlass(H-DD-WC) with $N_s^{\mathsf{frt}} \neq 0$ is used. The trend in WCL variation is similar to the trend when HourGlass(H-DD-NWC) is used. Figures 9.13a and 9.13b show the effect of increasing the number of HRT cores on WCL of HRT and FRT cores respectively. The WCL of both HRT and FRT cores increase with increase in number of HRT cores, as discussed for HourGlass(H-DD-NWC). From Figures 9.13c and 9.13d, I



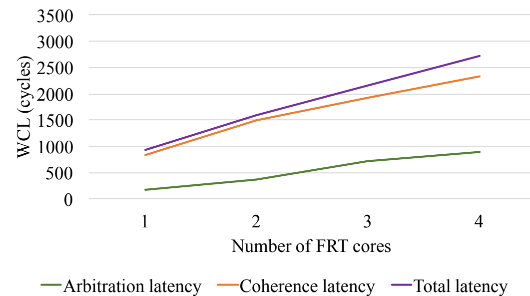
(a) Observed WCL for HRT cores.



(b) Observed WCL for FRT cores.



(c) Observed WCL for HRT cores.



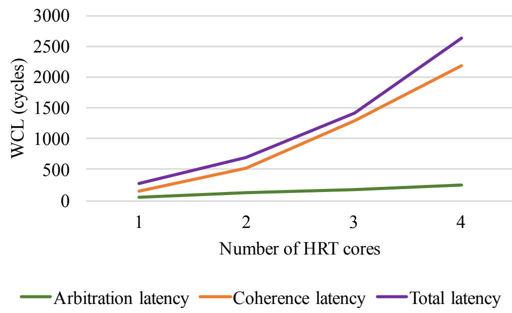
(d) Observed WCL for FRT cores.

Figure 9.13: Effect of scalability in HourGlass(H-DD-WC).

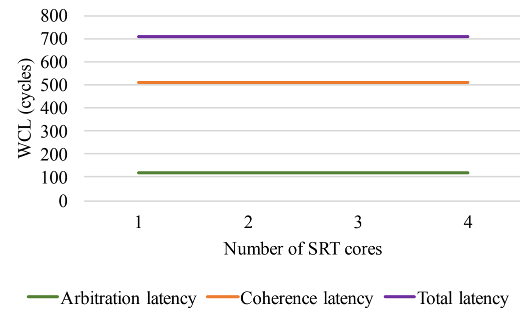
observe that WCL bound of FRT cores increase with increase in number of FRT cores. However, the WCL for HRT cores remain constant after the maximum number of FRT cores that can interfere with HRT cores is reached. This is given by the TDM period and the timer configuration for HRT cores.

9.6.3 Effect on HourGlass(H-DD-WC-0)

In HourGlass(H-DD-WC-0), no slots are allocated for SRT cores and hence the SRT cores do not have a guaranteed bound. Recall that the WCL of HRT cores is dependent on the number of HRT cores, and independent of the number of SRT cores, when this arbitration is used. In Figure 9.14a, the WCL of HRT cores increases with increase in the number of HRT cores. In particular, the arbitration latency is proportional to the TDM period, which is a function of number of HRT cores, and the coherence latency depends on the timer values, which in turn are fixed in terms of TDM periods. In addition, the coherence



(a) Observed WCL for HRT cores.



(b) Observed WCL for HRT cores.

Figure 9.14: Effect of scalability in HourGlass(H-DD-WC-0).

latency increases quadratically with increase in the number of HRT cores, and dominates the total coherence latency. However, from Figure 9.14b, the WCL of HRT cores does not increase with the number of SRT cores. However, increasing the number of SRT cores may affect the performance for SRT cores, as they contend for slack slots of the HRT cores.

Chapter 10

Conclusion and Future Work

In this thesis, I propose a predictable time-based cache coherence protocol, HourGlass. I also discuss criticality-aware bus arbitration schemes that can be used for HourGlass resulting in three versions of the protocol: HourGlass(H-DD-NWC), HourGlass(H-DD-WC) and HourGlass(H-DD-WC-0). HourGlass is also included with timer mechanisms to allow a core to hold valid cache lines in its private cache for a certain time duration, that is set initially. This helps to improve the performance of FRT or SRT cores, while still providing guarantees for HRT cores. The timer configurations are set based on the requirements of HRT and FRT or SRT tasks of the application. It provides a trade-off between the WCL bounds for HRT cores and performance of FRT and SRT cores. Based on the requirements, the criticality-aware bus arbitration scheme is chosen. SRT cores are not required to have any guaranteed bounds, and so H-DD-WC-0 can be used for applications that haave SRT tasks.

As a future work, HourGlass can be extended to support all three levels of time-critical tasks. HourGlass currently considers two levels of time-criticality where tasks are either HRT and FRT or HRT and SRT. SRT tasks do not require guaranteed WCL bounds and only focus on improved average-case performance. So the arbitration policy can be extended to three level, where all HRT tasks are allocated *dedicated* slots, FRT tasks are allocated few *dedicated* slots and also *slack* slots of HRT tasks, and no slots for requests from SRT tasks. The SRT tasks can thus use only the *slack* slots if there are no pending requests from HRT and FRT tasks. To this end, HourGlass requires modifications to ensure it responds differently to requests from three different criticality levels.

HourGlass incurs a hardware overhead of 133 bits per cache line. However, this is comparable to the overhead incurred by other time-based cache coherence protocols. Moreover,

if the timer values are aligned to TDM periods, then the overhead is reduced to 13 bits per cache line. HourGlass (all versions) performs better than the real-time approaches *uncache-all* and *uncache-shared* and experiences a 1.73× and 1.71× slowdown compared to the conventional unpredictable MSI and MESI cache coherence protocols for the SPLASH-2 benchmarks. HourGlass(H-DD-WC-0) performs better than the other versions and provides tighter WCL bounds for HRT cores, with no bounds for SRT cores. This is due to the presence of *slack* slots that are used by SRT cores. HourGlass(H-DD-NWC) and HourGlass(H-DD-WC) exhibit increase in total execution times compared to the state-of-the-art real-time cache coherence protocol PMSI for SPLASH-2 benchmarks. This is due to the fact that SPLASH-2 do not have frequent shared data reuse and hence the use of timers are detrimental. HourGlass(H-DD-WC-0) performs slightly better than PMSI due to the use of *slack* slots. HourGlass is a predictable cache coherence protocol designed for mixed-time-criticality systems and it is useful where data is shared across different criticality levels. Most current work disable sharing of data across different criticality levels due to interference of FRT or SRT tasks on HRT tasks leading to unpredictability. However, HourGlass is a predictable approach that provides improved average-case performance for FRT or SRT cores, while still guaranteeing WCL bounds for HRT cores.

References

[1] Autobench 2.0 - performance suite for multicore automotive processors. http://www.eembc.org/autobench2/index.php.

[2] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. Comparison of hardware and software cache coherence schemes. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, 1991.

[3] ARM. Cortex-R5 and Cortex-R5F Technical Reference Manual. 2011.

[4] ARM. ARM Architecture Reference Manual ARMv8. 2013.

[5] Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 13–22. IEEE, 2010.

[6] Sanjoy K Baruah, Alan Burns, and Robert I Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 34–43. IEEE, 2011.

[7] Nathan Binkert and et al. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 2011.

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

[9] J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *21st Euromicro Conference on Real-Time Systems (ECRTS)*, 2009.

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

[11] B. Cilku, A. Crespo, P. Puschner, J. Coronel, and S. Peiro. A TDMA-based arbitration scheme for mixed-criticality multicore platforms. In *2015 International Conference on Event-based Control, Communication, and Signal Processing (EBCCSP)*, June 2015.

[12] B. Cilku, B. Frmel, and P. Puschner. A dual-layer bus arbiter for mixed-criticality systems with hypervisors. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, July 2014.

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

[14] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–15. IEEE, 2013.

[15] Manil Dev Gomony, Jamie Garside, Benny Akesson, Neil Audsley, and Kees Goossens. A globally arbitrated memory tree for mixed-time-criticality systems. *IEEE Transactions on Computers*, 66(2):212–225, 2017.

[16] Giovani Gracioli and Antônio Augusto Fröhlich. On the design and evaluation of a real-time operating system for cache-coherent multicore architectures. *SIGOPS Oper. Syst. Rev.*, 2016.

[17] Vincent Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *ACM SIGPLAN Notices*, volume 50, pages 1–10. ACM, 2015.

[18] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *30th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

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

[20] M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling dram memory accesses for multi-core mixed-time critical systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[21] Mohamed Hassan, Anirudh Kaushik, and Hiren Patel. Predictable cache coherence for multi-core real time systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[22] Intel. Intel 64 and IA-32 architectures software developers manual. *Volume 3A: System Programming Guide, Part*, 1(64), 64.

[23] Leonidas I Kontothanassis and Michael L Scott. Software cache coherence for large scale multiprocessors. In *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pages 286–295. IEEE, 1995.

[24] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared data caches conflicts reduction for WCET computation in multi-core architectures. In *18th International Conference on Real-Time and Network Systems (RTNS)*, 2010.

[25] Marco Paolieri and et al. Hardware support for WCET analysis of hard real-time multicore systems. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[26] 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 *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, 2014.

[27] Keun Sup Shim, Myong Hyon Cho, Mieszko Lis, Omer Khan, and Srinivas Devadas. Library cache coherence. 2011.

[28] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. Cache coherence for gpu architectures. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 578–590. IEEE, 2013.

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

[30] Sanket Tavarageri, Wooil Kim, Josep Torrellas, and P Sadayappan. Compiler support for software cache coherence. In *High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on*, pages 341–350. IEEE, 2016.

[31] S. C. Woo and et al. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings 22nd Annual International Symposium on Computer Architecture (ISCA)*, 1995.

[32] Xiangyao Yu and Srinivas Devadas. Tardis: Time traveling coherence algorithm for distributed shared memory. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 227–240. IEEE, 2015.

[33] H. Yun, R. Pellizzon, and P. K. Valsan. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *27th Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[34] Qingling Zhao, Zonghua Gu, and Haibo Zeng. Hlc-pcp: A resource synchronization protocol for certifiable mixed criticality scheduling. *IEEE Embedded Systems Letters*, 6(1):8–11, 2014.