Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems

Anirudh Mohan Kaushik, *Member, IEEE*,
Mohamed Hassan, *Member, IEEE*, and Hiren Patel, *Member, IEEE*

Abstract—This article addresses the challenge of allowing simultaneous and predictable accesses to shared data on multi-core systems. We propose a collection of predictable cache coherence protocols, which mandate the use of certain design invariants to ensure predictability. In particular, we enforce these invariants by augmenting the classic modify-share-invalid (MSI) protocol and modify-exclusive-share-invalid (MESI) protocol. This allows us to derive worst-case latency bounds on the resulting predictable MSI (PMSI) and predictable MESI (PMESI) protocols. Our analysis shows that while the arbitration latency scales linearly, the coherence latency scales quadratically with the number of cores, which emphasizes the importance of accounting for cache coherence effects on latency bounds. We implement PMSI and PMESI in a detailed micro-architectural simulator, and execute SPLASH-2 and synthetic workloads. Results show that our approach is always within the analytical worst-case latency bounds, and that PMSI and PMESI improve average-case performance by up to 4× over cache bypassing mechanisms that disallow caching of shared data in the cores' private caches. PMSI and PMESI have average slowdowns of 1.45× and 1.46× compared to conventional MSI and MESI protocols, respectively.

Index Terms—Multi-core real time systems, cache coherence, predictability

1 Introduction

In HARD real-time systems, correctness depends on both the functioning behavior, and on the timing of that behavior [1]. Applications running on these systems have strict requirements on meeting their execution time deadlines. Missing a deadline in a hard real-time system may cause catastrophic failures [2]. Therefore, ensuring that deadlines are always met via static timing analysis is mandatory for such systems. Timing analysis computes an upper bound for the execution time of each running application on the system by carefully accounting for hardware implementation details, and using sophisticated abstraction techniques. The worst-case execution time (WCET) of any application has to be less than or equal to this upper bound to achieve predictability. As application demands continue to increase from the avionics [3] and automotive [4] domains, there is increasing attention in deploying multi-core platforms. This is primarily due to the benefits multi-core platforms provide in cost, and performance. However, multi-core platforms pose new challenges towards guaranteeing temporal requirements of running applications. Among these challenges, achieving predictable shared data accesses in real-time applications has gained recent attention from the research community [5], [6], [7], [8], [9]. Recent work has showed clear evidence of data sharing between real-time tasks in practical real-time domains deployed on multi-core platforms [6], thereby making this an important challenge and the main focus of this work.

One mechanism for managing shared data accesses that is standard in existing multi-core platforms is *cache coherence* [10], [11]. Cache coherence mechanism provides all cores in the multi-core platform access to coherent data that may be cached in their private caches [11]. Cache coherence is realized by implementing a protocol that specifies a core's activity (read or write) on cached shared data based on the activity of other cores on the same shared data. While cache coherence can be implemented in software or hardware, modern multi-core platforms implement the cache coherence protocol in hardware [10]. This is so that software programmers do not have to explicitly manage coherence of shared data in the application. A recent work studied the effect of cache coherence on execution time using different Intel and AMD processors and coherence protocols [12]. The study compared execution times between executing an application sequentially and in parallel. It concluded that the interference from cache coherence can severely reduce benefits gained from parallelism. In fact, it can make parallel execution 3.87× slower than sequential execution [12]. For real-time applications that share data, this emphasizes the importance of considering cache coherence effects when deriving WCET bounds. However, as observed by a recent survey [13], there is no existing technique to account for the effects of cache coherence in static timing analysis in real-time systems.

Current techniques, which do not use cache coherence, enable coherent data sharing by enforcing restrictions on shared data accesses. These techniques include (1) disabling

- *Anirudh Mohan Kaushik and Hiren Patel are with the Electrical and Computer Engineering, University of Waterloo, 8430, Waterloo, ON N2L 3G1, Canada. E-mail: {amkaushi, hiren.patel}@uwaterloo.ca.*
- *Mohamed Hassan is with the Electrical and Computer Engineering, McMaster University, 3710, Hamilton, ON L8S 4L8, Canada. E-mail: mohamed.hassan@mcmaster.ca.*

Manuscript received 29 Feb. 2020; revised 5 Oct. 2020; accepted 10 Oct. 2020. Date of publication 12 Nov. 2020; date of current version 8 Nov. 2021. (Corresponding author: Anirudh Mohan Kaushik.) Recommended for acceptance by H. Jiang. Digital Object Identifier no. 10.1109/TC.2020.3037747

caching of shared data [7], [9], (2) mapping tasks that share data to execute on the same core [5], [14], [15], and (3) marking shared data accesses as critical sections such that they are accessed by a single core at any time instance [16]. These techniques enable predictable and coherent data sharing at the expense of (1) severely degrading average-case performance, (2) imposing task scheduling restrictions, and (3) application and real-time operating system (RTOS) modifications. On the other hand, a predictable hardware cache coherence mechanism can address these three limitations of current techniques as it (1) allows shared data to reside in the private caches of multiple cores simultaneously, (2) does not impose task scheduling restrictions, and (3) does not require application modifications.

In this work, we take the first steps towards using hardware cache coherence for managing predictable shared data accesses. A key contribution of this work is to show that a simple combination of a conventional hardware cache coherence protocol and a shared bus that deploys a predictable shared bus arbitration policy is *insufficient* to guarantee predictable shared data accesses under cache coherence. We address the problem of maintaining cache coherence in multi-core real-time systems by analyzing and modifying conventional hardware cache coherence protocols. The resulting cache coherence protocols allow for predictable and coherent data sharing in a manner amenable for timing analysis [17]. We extend our previous work [17] by analyzing the Modified-Exclusive-Shared-Invalidate (MESI) protocol, which offers additional average-case performance benefits over the MSI protocol [11]. We then design the predictable MESI (PMESI) protocol, and extend the timing analysis to derive the worst-case latency (WCL) of a memory request under PMESI. Our timing analysis shows that although PMESI has additional performance benefits over PMSI, the WCL of a memory request under PMSI and PMESI are the same. We also identify opportunities to improve the average-case performance of PMESI through additional hardware modifications, which results in a new protocol Opt-PMESI.

In summary, this work extends our previous work [17], and makes the following contributions:

1) We analyze the conventional MESI coherence protocol, and highlight the unpredictable behaviors in this protocol. We propose extensions to the MESI coherence protocol to guarantee predictability resulting in the predicable MESI protocol. The PMESI protocol satisfies the design invariants for predictability (Section 5) through protocol changes and architectural extensions (Section 6). We also design the Opt-PMESI protocol that improves average-case performance over PMESI through hardware optimizations.
2) We provide a timing analysis for our proposed coherence protocols and decompose the analysis to highlight the contributions to latency due to arbitration logic and communication of coherence messages between cores (Section 7).
3) We evaluate the proposed coherence protocol using the gem5 simulator [18] (Section 8). Performance evaluation using synthetic and SPLASH-2 workloads shows that PMSI, PMESI, and Opt-PMESI achieve up to $4\times$ speedup over competitive predictable approaches for a quad-core system while guaranteeing predictability. Furthermore, Opt-PMESI improves performance over PMSI and PMESI by up to 12 percent.

2 Related Work

Prior research efforts investigated the access latency overhead resulting from shared buses [2], caches [8], [19], [20], and dynamic random access memories (DRAMs) [21], [22], [23]. For shared caches, most of these efforts primarily focused on preventing a task's data accesses from affecting another task's data accesses. They used data isolation between tasks by utilizing strict cache partitioning [8] or locking mechanisms [19]. Authors in [20] promoted splitting the data cache into multiple data regions that simplified the analysis. However, they indicated that cache coherence is still an issue that has to be addressed. Similarly, several proposals for shared main memories deployed data isolation that assigned a private memory bank per core [21], [22]. However, we find that data isolation suffers from three limitations. The first limitation is that it disallows sharing of data between tasks; thus, disabling any communication across applications or threads of parallel tasks running on different cores. The second limitation is that it may result in poor memory or cache utilization. For instance, a task may keep evicting its cache lines if it reaches the maximum of its partition size, while other partitions may remain underutilized. The third limitation is that it does not scale with increasing number of cores. For example, the number of cores in the system has to be less than or equal to the number of DRAM banks to be able to achieve isolation at DRAM.

Recent works [23], [24] recognized these limitations, and offered solutions for sharing data. Authors in [23] shared the whole memory space between tasks for main memory, and [24] suggested a compromise that divided the memory space into private and shared segments for caches. Nonetheless, these approaches focused on the impact of sharing memory on timing analysis, and they did not address the problem of data correctness resulting from sharing memory. Authors of [25] studied the overhead effects of co-running applications on the timing behavior in the avionics domain, where cache coherence was one of the overhead sources. A recent survey [13] observed that there is no existing technique to include the effects of data coherence on timing analysis for multi-core real-time systems.

However, there exist approaches that attempt to eliminate unpredictable scenarios that arise from data sharing. Authors in [14] proposed data sharing-aware scheduling policies that avoided running tasks with shared data simultaneously. A similar approach proposed by [12], [15] redesigned the real-time operating system to include cache partitioning, task scheduling, and feedback from the performance counters to account for cache coherence in task scheduling decisions. Such approaches rely on hardware counters that feed the schedule with information about memory requests. They also require modifications to existing task scheduling techniques. For example, the solution in [14] is not adequate for partitioned scheduling mechanisms. A different solution introduced in [16] applied source-code modifications to mark instructions with shared

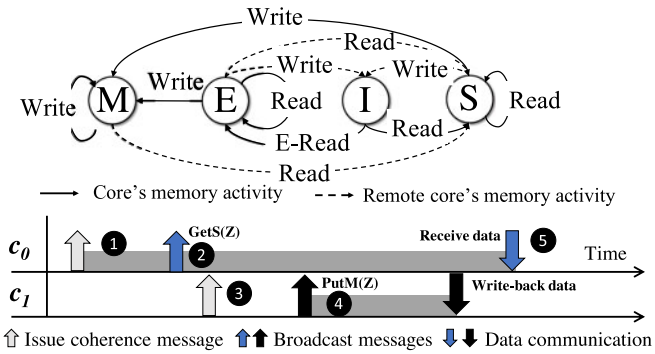


Fig. 1. MESI coherence protocol and illustrative example.

data as critical sections. These critical sections were protected by locking mechanisms such that they were accessed only by a single core at any time instance. This solution suffers from two limitations. The first limitation is that the software is responsible for maintaining cache coherence to guarantee shared data correctness. As a result, additional changes to the software are necessary in order to explicitly manage cache coherence. The second limitation is that only one core can access a cache line of shared data at a time. Other cores requesting this data must wait until a core completes all operations on the shared data. In the worst case, this is equivalent to sequential execution. On the other hand, our proposed cache coherence protocols (PMSI and PMESI) allow tasks to simultaneously access shared data, which considerably improves performance. In addition, PMSI and PMESI do not pose any requirements on task scheduling techniques, and they do not require software modifications.

3 Cache Coherence Background

The objective of cache coherence is to provide all cores read access to the most recent write on shared data. Incoherent sharing of data occurs when multiple cores read different versions of the same data that is present in their private cache hierarchies. A coherence protocol avoids data incoherence by deploying a set of rules to ensure that cores access the correct version of data at all times. Typically, the coherence protocol maintains data coherence at cache line granularity, which is a fixed size collection of data. A state machine implements these rules with a set of states representing read and write permissions on the cache line, and transitions between states are triggered based on the activity of cores on the shared data. The cache controller typically implements the coherence protocol. General purpose processors deploy different variants of coherence protocols.

Cache coherence protocols deployed in current multi-core platforms consist of three fundamental stable states, which establish the MSI protocol: *modified* (M), *shared* (S), and *invalid* (I) [11]. A cache line in *modified state* means that the current core has written to it and it did not propagate the updated data to the shared memory yet. Only one core can have a specific cache line in a modified state, and is referred to as the *owner*. A cache line in *shared state* means that the core has a valid, yet unmodified version of that line. One or more cores can have versions of the same cache line in shared state to allow for fast read accesses. Cores that have the same cache line in the shared state are referred to as *sharers*. This constraint of one owner for a cache line or multiple cores sharing a cache line is referred to as the *single-writer multiple-reader* (SWMR) invariant [11]. A cache line in *invalid state* denotes the unavailability of that line in the cache or that its data is outdated and no longer valid.

The MESI protocol introduces the exclusive (E) state to the MSI protocol as shown in Fig. 1. The additional E state is an optimization to the MSI protocol, and features in cache coherence protocols deployed in current multi-core platforms. A core that receives a line in E state from the shared memory (E-Read in Fig. 1) guarantees that no other core has the same line in a valid state (S or M states). To enable this optimization, the shared memory keeps track of additional states to identify (1) a read-only copy of the line present in cores' private cache (S state), (2) no copy of the line present in any cores' private cache (I state),[1] and (3) an exclusive or modified copy of the line is present in a core's private cache (E/M state). The E state allows for one performance optimization for store requests. A core that has a line in the E state can complete a store request without issuing any coherence messages on the bus. This is because a core that has a line in the E state implies that there are no other cores that have the same line in their private caches. Hence, no private copies of the line need to be invalidated. We refer to this performance optimization as *silent stores*. On the other hand, in MSI, a core performing a store operation on a line in S or I state must issue coherence messages on the shared bus before it can complete its store operation.[2]

A cache controller changes the state of the cache line by observing the bus for coherence messages related to the same cache line by other cores, known as *bus snooping cache coherence* or receiving action messages from a centralized shared cache controller, known as *directory-based cache coherence*. In this work, we focus on bus snooping cache coherence as it is typically implemented in multi-core platforms with a small number of cores, which is the case in current real-time systems. For example, bus snooping is adopted in ARM chips such as [26]. For bus snooping protocols, we distinguish between two types of messages: *coherence messages* and *data messages*. We define *coherence messages* as messages that represent an action corresponding to a core's activity on the cache line, and *data messages* represent data sent or received by a core as a consequence of a core's activity.

We provide an example to illustrate cache coherence using Fig. 1. Consider the following scenario: core c_0 *issues* a load to cache line Z, and c_1 has Z in modified (M) state. The load to Z first checks c_0's private cache for its existence. On a private cache hit, the necessary data is supplied, and the load is marked complete. Private cache hits do not generate coherence activity. On a cache miss, the private cache controller of c_0 *generates* a coherence message of the form $\mathsf{Get}(\mathsf{A})$. If a core issues a load request to Z, its cache

1. Note that for MSI cache coherence protocol, the shared memory combines these two states into one state (I state) [11].

2. In this work, we do not consider the MOESI and MESIF cache coherence protocols. The owner (O) and forwarding (F) states are optimizations when cores communicate with each other directly without shared memory involvement. In this work, we consider multi-core configurations where all communication between cores are through the shared memory (Section 4).

controller generates a GetS(Z) message. If a core issues a store request to Z, its cache controller generates a GetM(Z) message. In Fig. 1, c_0 issues a GetS(Z) at ❶. If Z is marked for eviction, and is in M state, the core first generates a PutM(Z) message, and then writes Z to the shared memory. This operation is called a write-back operation. If the core has Z in a shared state and wants to modify it, it generates an Upg(Z) message. The cache controller then *broadcasts* the GetS(Z)/GetM(Z)/PutM(Z)/Upg(Z) message on the snoopy bus. c_0 broadcasts its GetS(Z) on the snoopy bus at ❷. A message is said to be *ordered* on the bus when all cores and the shared memory observe the memory request on the bus. A core observes its own messages on the bus as well as messages by other cores. We refer to the former as Own, and to the latter as Other. At ❸, c_1 observes c_0's GetS(Z) as an OtherGetS(Z), and marks Z for write-back. At ❹, c_1 broadcasts a PutM(Z) and completes the write-back of Z. The shared memory sends the updated Z to c_0, and c_0 completes its load request to Z at ❺ on receiving the requested data.

Transient states capture state changes to a core's cache line due to intervening events while the core's memory request to the cache line is pending [11]. A core with a pending memory request to a cache line can observe memory requests to the same cache line from other cores due to non-atomic reordering buses, which are standard in multi-core platforms due to their performance benefits [11]. We categorize the transient states into two categories:

*Transient States for Coherence Messages.*These transient states denote that a core is waiting for its own coherence message to be placed on the bus. A core's own coherence message may be delayed on the bus due to the presence of other messages on the bus. For instance, when a core c_i issues a load request to an invalid line, it broadcasts a GetS() message. Because of the non-atomicity and reordering nature of the bus, c_i might receive its requested data before it observes its message on the bus as shown by [11].

*Transient States for Data Messages.*These states denote that a core is waiting for data either from a core that has the data in its private cache hierarchy or from the shared memory.

4 System Model

We consider a multi-core system with N cores, $\{c_0,c_1,...,c_{N-1}\}$. Each core has a private cache, and all cores have access to a shared memory. This shared memory can be an on-chip last-level cache (LLC), an off-chip DRAM, or both. Tasks running on cores share data. These tasks can belong to a parallel application that is distributed across cores, or different applications that communicate between each other. Cores can share the whole shared memory space similar to [23] or share part of the memory space similar to [24]. We do not impose any restrictions on how the interference on the shared memory is resolved, whether it is the LLC or the DRAM. Furthermore, we do not require any special demands from the task scheduling mechanism. This allows one to integrate the proposed solution to current task scheduling techniques, and to various mechanisms that control accesses to shared memories in multi-core real-time systems. Cores share a common snooping bus that connects private caches of cores to the shared memory. This shared bus allows for cores to broadcast their memory requests to other cores and the shared memory, and data transfers between the shared memory and cores. The shared bus also transfers coherence messages deployed by the coherence protocol to ensure data correctness. Cores *snoop* the bus to observe memory activity of other cores. The system deploys a predictable arbitration on the shared bus. Note that data transfers between private caches are only via the shared memory (no cache-to-cache transfers). Although some of the problems addressed in this paper may also apply to systems supporting cache-to-cache transfer, those systems are not the focus of this paper. The proposed solution is independent of the core architecture, and the predictable arbitration mechanism on the bus. However, the analysis and experiments we present in this work consider a system with in-order cores, and a time-division-multiplexing (TDM) bus as the base arbitration scheme. A TDM slot width allows for one data transfer between shared memory and the private cache including the overhead of necessary coherence messages.

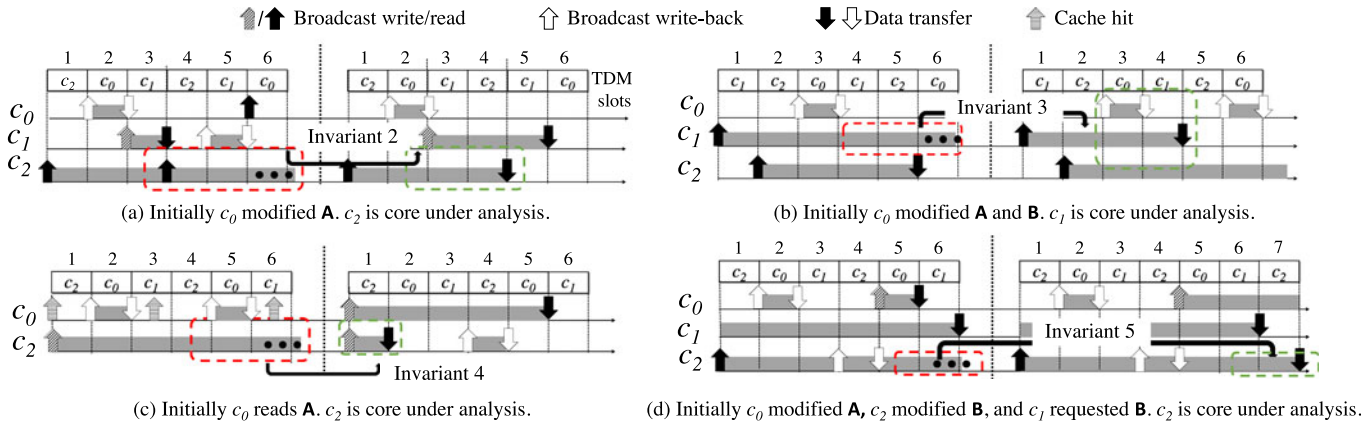
5 Invariants for Predictable Coherence

A cache coherence protocol ensures correctness of shared data across all cores in a multi-core platform. As we show in this section, simply adopting a predictable arbiter in this case does not necessarily mean that tasks will have predictable latencies upon accessing the shared memory. This is because, as illustrated in Section 3, the latency suffered by one core accessing a shared line is dependent on the coherence state of that line in the private caches of other cores. Two major contributions of this paper are (1) to identify these unpredictable scenarios, and (2) to propose invariants to address them. In this section, we describe these unpredictable scenarios, and propose design invariants to address these scenarios. Exact sources of unpredictability in current multi-core platforms are dependent on the cache coherence protocol and micro-architecture details of the cache controllers, which are proprietary and are not publicly available. The proposed invariants are general design guidelines, which are independent of the adopted cache coherence protocol implementation and the underlying platform architecture.

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

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

Investigating the implications of a conventional coherence protocol on the WCET, we find that there are *five* major sources that can lead to unpredictable behavior. We group



(a) Initially c_0 modified **A**. c_2 is core under analysis.

(b) Initially c_0 modified **A** and **B**. c_1 is core under analysis.

(c) Initially c_0 reads **A**. c_2 is core under analysis.

(d) Initially c_0 modified **A,** c_2 modified **B**, and c_1 requested **B**. c_2 is core under analysis.

Fig. 2. Unpredictability scenarios and corresponding invariants that fix the unpredictable scenarios.

these sources into two categories: *inter-core interference* and *intra-core interference*. Figs. 2a, 2b, 2c, and 2d illustrate example scenarios for these sources. The example scenarios consider a system with three cores, c_0, c_1, and c_2, and deploys a TDM arbitration across cores. If the request type is not specified whether it is a load or store, that means the scenario is agnostic to it. Each of Figs. 2a, 2b, 2c, and 2d separately defines the initial system state and the core under analysis for the corresponding scenario. We denote TDM slot i as ❶.

5.1 Inter-Core Coherence Interference

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

5.1.1 Interference on Same Line

The first source of unpredictability arises from multiple cores reading the same modified cache line, say A. If a core requests to modify A, it has to wait for the owner to write-back A to the shared memory. In Fig. 2a, initially, c_0 has a modified version of A in its private cache. The core under analysis is c_2. At ❶, c_2 broadcasts a load request to A. Since c_0 has the modified version of A, it has to write-back the updated A to the shared memory first. However, this is c_2's slot; thus, c_0 has to wait for its allocated slot to perform the write-back. Hence, at ❷, c_0 writes back A to the shared memory in its slot. At ❸, c_1 broadcasts a store request to A. Since the shared memory has the updated version of A, c_1 is able to obtain A and modify it. As a result, c_2 re-broadcasts a load request to A at ❹. This time c_2 has to wait for c_1 to write-back A. From c_2's perspective, the events at ❹ are a repetition of the events at ❶, c_2 re-broadcasts its request to A and waits for another core to write it back. Thus, this situation is repeatable and can result in unbounded memory latency. Although c_2 is granted access to the bus, it is unable to obtain the requested data due to the coherence interference.

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

Proposed Solution. Invariant 2 requires memory to service requests to the same cache line in their *arrival order*; thus, it guarantees that a line being requested by a core will not be invalidated before the core accesses it. In the above example, the memory serviced requests based on the arbitration schedule order, which is different from the arrival order resulting in unbounded memory latency. Imposing Invariant 2 in Fig. 2a, c_2's request to A arrives to the shared memory before c_1's request; therefore, c_1 has to wait for c_2 to execute its operation before it gains an access to A. Note that in conventional snooping bus-based coherence protocols, this invariant is realized by ensuring that the shared memory responds to memory requests from cores based on their broadcasted order [11].

5.1.2 Interference on Different Lines

The second source of interference arises when multiple cores request different cache lines that are modified by the same core (owner). As a result, the owner has to write-back the modified lines requested by the other cores to the shared memory. For instance in Fig. 2b, c_0 has modified versions of lines A and B. The core under analysis is c_1. c_1 broadcasts a request to A in ❶, and c_2 broadcasts a request to B in ❷. Accordingly, c_0 has to write-back both A and B to the shared memory. Since c_0 can schedule one memory transfer in a slot, it can write-back only one line to the shared memory. If no predictable mechanism manages the write-backs, c_0 can pick any pending one. At ❸, c_0 writes back B. Therefore, at ❹, c_1 is stalled on A. This situation can repeat indefinitely. While c_1 is waiting for A, c_2 can ask for another line, which is also modified by c_0 and the same situation can repeat.

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

Proposed Solution. Invariant 3 imposes an order in servicing coherence messages from other cores (write-backs, for example). The right side of Fig. 2b deploys Invariant 3. Since the request to A arrives before that to B, c_0 has to write-back A first (in ❸) then B (in ❻); thus, a predictable behavior is guaranteed.

5.1.3 Writes to Non-Modified Lines

The third source is due to write hits in the private cache to non-modified lines. Recall that the predictable bus arbiter only controls accesses to the shared bus. As a result, a

request that results in a hit in the private cache can proceed without waiting for the corresponding core slot. However, store requests to unmodified lines that hit in the private cache can result in the following two unpredictable scenarios described in Figs. 2c and 2d.

The first scenario arises when multiple cores update the same line and one of the cores has the line in an unmodified state in its private cache. For example, in Fig. 2c, c_0 has a version of A in its private cache that is not modified. At ❶, c_2 broadcasts a load request to A, while simultaneously c_0 has a write operation to A that results in a hit in its private cache. Consider the scenario where c_0's write hit on A occurs first. As a result, c_2 has to wait until c_0 writes back A. This scenario is shown in Fig. 2c. After c_0 writes back A in ❷, c_0 again has another write hit to A in ❸. Again, c_2 has to wait for c_0 to write-back A. Consequently, this situation is repeatable and can starve c_2.

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

Proposed Solution. Invariant 4 stalls a store request by a core, which is a hit to a non-modified line until the arbiter grants an access slot to that core. Thereby, it avoids the aforementioned unpredictable consequences. It is worth noting that Invariant 4 aligns with Invariant 1 as follows. Invariant 1 mandates that a core can initiate coherence messages into the bus only when it is granted an access to it by the arbiter. Although a write hit to a non-modified line does not need data from the shared memory, it still needs to send coherence messages on the bus. This is necessary to invalidate local copies of the same line that other cores have in their private caches. Accordingly, a write hit to a non-modified line has to wait for a granted access by the arbiter. On maintaining Invariant 4 in Fig. 2c, the following behavior is guaranteed. Since ❶ belongs to c_2, and c_0's request is a write hit to A, which is not modified, c_0 must wait for its slot to that request. c_2 broadcasts its store request to A in ❶, and c_0 invalidates its own local copy of A. Since no core has a modified version of A, c_2 obtains A from the shared memory and performs the write operation.

Invariant 4 resolves the race situation between a request generated by a core in its designated slot and write hits from other cores. However, a second unpredictable scenario is possible that Invariant 4 does not manage. We describe this scenario using Fig. 2d. Initially, c_0 has a modified version of A, c_2 has a modified version of B, and c_1 has requested B. At ❶, c_2 broadcasts a load request to A; thus, c_0 updates the shared memory with the modified value of A at ❷. Since c_2's request is a load, c_0 does not invalidate its local version of A. At ❹, c_2 has two pending actions: fetching A from memory, and writing back B to the memory in response to c_1's request. Assume that c_2 chooses to write-back B. Therefore, its request to A waits for the next slot. At ❺, c_0 has a write hit to A. Consequently, since this is c_0's slot, it conforms with Invariant 4; thereby, it modifies A. At ❻, c_2 has to re-broadcast its request to A and wait for c_0 to write-back A to memory again. From c_2's perspective, this situation is similar to the situation at ❶. Similarly, in subsequent periods, after c_0 writes back A, it can have a write hit to A before c_2 receives it from the memory. Clearly, this situation is repeatable indefinitely, and creates unbounded memory latency for c_2.

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

Proposed Solution. Invariant 5 stalls a store request to a non-modified line until all pending requests from previous slots are completed. Thereby, it avoids the above unpredictable scenario. Maintaining Invariant 5 in the right side of Fig. 2d, the following behavior is guaranteed. During c_0's slot, it has a hit to A. Since A is non-modified by c_0 and is previously requested by c_2, the write hit cannot be processed. Accordingly, c_2 obtains A from the shared memory in its next slot and performs its operation. c_0's request to A is broadcasted afterwards in the corresponding slot.

5.2 Intra-Core Coherence Interference

Intra-core coherence interference arises due to multiple memory activity from the *same* core such as a core's own pending request and its response to a request from another core. This response is for example, a write-back to a line that this core has in a modified state. For example, consider a time instance where a core c_0 has a pending request to A and a pending write-back response on B due to another core's (c_1) request on B. Both the pending request and response requires access to the shared bus. In c_0's allocated slot, c_0 broadcasts the request to A. Hence, the write-back response on A must wait for the next allocated slot. However, in the next allocated slot, c_0 may broadcast a pending request to a different cache line (C). In this way, the write-back to A can indefinitely stall, which results in unbounded latency for c_1's request.

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

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

6 Predictable Coherence Protocols

We show the effectiveness of the proposed invariants by applying them to the conventional MSI and MESI protocols. This results in predictable MSI (PMSI) and predictable MESI protocols for multi-core real-time systems. To ensure that the invariants described in Section 5 are held, we propose architectural modifications and additional coherence states. The architectural modifications apply to both PMSI and PMESI, and the additional coherence states are protocol specific.

Invariants require either architectural modifications or a combination of both architectural modifications and additional coherence states. For example, Invariants 1 and 2 require only architectural modifications and no changes to the coherence protocols. On the other hand, Invariants 3, 4, 5, and 6 require modifications to both the architecture and the coherence protocol. This is because Invariants 3 and 6

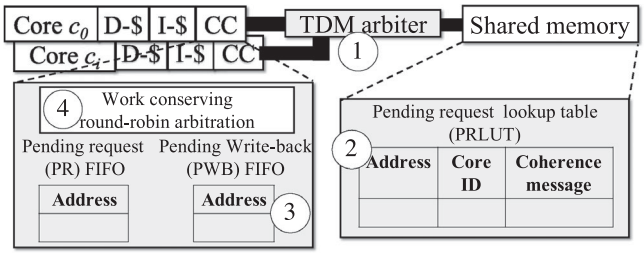


Fig. 3. Architectural changes necessary for PMSI and PMESI.

regulate the write-back operation of cache lines. Since a core has to wait for a designated write-back slot to write-back a cache line A, it has to maintain A in a transient state to indicate that A is waiting for write-back. Similarly, Invariants 4 and 5 regulate the store hit operation to non-modified lines. A core has to wait for a designated slot to perform the store hit operation to a cache line, say B. Accordingly, it has to maintain B in a transient state indicating that it has a pending store to B.

In the following sections, we describe the architectural modifications that are required for both PMSI and PMESI coherence protocols (Section 6.1), and then describe the PMSI and PMESI protocol modifications (Section 6.2.3).

6.1 Architectural Modifications

Fig. 3 depicts a multi-core system with a private cache for each core and a shared memory connected to all cores via a shared bus. A TDM bus arbiter manages accesses to the shared memory. The proposed architecture changes are highlighted in grey.

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

These architectural changes, along with the coherence protocol changes, also satisfy Invariants 4 and 5 as follows. If a core c_i has a store hit to a non-modified line A, it has to broadcast an Upg() coherence message on the bus. With ①, the arbiter does not allow this Upg() message on the bus unless it is the TDM slot of the initiating core. In consequence, the store hit to A is postponed to c_i's next slot, which implements Invariant 4. Assume that during c_i's next slot, there were one or more pending requests to A from other cores that arrived before c_i's request. According to Invariant 5, c_i's store hit to A has to wait until these pending requests are serviced. Recall that PR LUT ② queues pending requests. If the store hit is to one of these lines, the arbiter does not select the store hit to execute during this slot. Accordingly, Invariant 5 is fulfilled.

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

6.2 Protocol Modifications

We discuss the protocol modifications to the MSI and MESI protocols that work in tandem with the hardware structures described earlier. The protocol modifications result in new protocols: PMSI and PMESI protocols respectively. We also describe an optimized variant of PMESI, Opt-PMESI, which adds hardware and protocol extensions to improve the average-case performance of PMESI. We first describe the transient states that are removed and unmodified across all the protocols (Sections 6.2.1 and 6.2.2), and then introduce new transient states introduced for each protocol in Sections 6.2.3, 6.2.4, and 6.2.5. Table 1 shows the private caches' coherence states for a cache line and the transitions between these states for the PMSI protocol. We do not make changes to the coherence states for the shared memory, and hence it is not shown. Shaded cells represent transitions that are not possible under correct operation. Cells marked with "-" represent situations where no transition occurs, and the coherence state remains unchanged.

6.2.1 Removed Transient States

For a real-time system, transient states that indicate unavailability of cache line in the private caches and waiting for coherence messages to appear on the bus are not needed. Examples of such transient states include IM^a and IS^a (Section 3). On deploying a predictable bus arbitration, once a core is granted access to the bus, no other core can issue a coherence message during that slot. This is assured by Invariant 1. Accordingly, during a core's slot, its coherence messages are not disrupted by messages from other cores. By removing these transient states, PMSI, PMESI, and Opt-PMESI has fewer states and transitions compared to their respective conventional protocols [11].

6.2.2 Unmodified Transient States

Transient states that denote the waiting for data response are retained. Examples of such transient states are IS^d and IM^d that denote a core's read or write request is waiting for data respectively. This is because if c_i issues a request to a cache line that is modified by another core c_j, c_i must wait until c_j writes back that cache line to the shared memory. Accordingly, c_i has to move to a transient state indicating that it is waiting for a data response from the memory. In addition, there are three other unmodified transient states such as IS^dI, IM^dI, and IM^dS. These states indicate that the

TABLE 1
Private Memory States for PMSI, PMESI, and Opt-PMESI

	Core events			Bus events						
	Load	Store	Replacement	OwnData	OwnUpg	OwnPutM	OtherGetS	OtherGetM	OtherUpg	OtherPutM
I	Issue GetS/IS^d	Issue GetM/IM^d					—	—	—	—
S	Hit	Issue Upg/**SM^w**	I				—	I	I	
M	Hit	Hit	Issue PutM/**MI^{wb}**				Issue PutM/**MS^{wb}**	Issue PutM/**MI^{wb}**		
IS^d				If E-Read/E, else Read/S			—	IS^dI	IS^dI	—
IM^d				Write/M			IM^dS	IM^dI		—
SM^w			Stall		**Write/M**		—	Reissue store/**I**	Reissue store/**I**	
MI^{wb}	**Hit**	**Hit**	—			Send data to memory/**I**	—	—		
MS^{wb}	**Hit**	**hit**	**MI^{wb}**			Send data to memory/**S**	—	**MI^{wb}**		
IM^dI				Write/**MI^{wb}**			—	—		—
IS^dI				Read/I			—	—	—	—
IM^dS				Write & Issue PutM/**MS^{wb}**			—	IM^dI		—
E	Hit	Hit/M	(A) Issue PutM/**EI^{wb}** (B) Send NoData to memory/I				(A) Issue PutM/**ES^{wb}** (B) Send NoData to memory/S	(A) Issue PutM/**EI^{wb}** (B) Send NoData to memory/I		
EI^{wb}	**Hit**	**Hit**/MI^{wb}	—			Send data to memory/**I**	—	—		
ES^{wb}	**Hit**	**Hit**/MS^{wb}	**EI^{wb}**			Send data to memory/**S**	—	**EI^{wb}**		

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

core has to take an action after receiving the data and perform the operation. For example, the transient state IS^dI indicates that a core waiting on data for a broadcasted load operation observed a remote store operation. The core on receiving the data completes the load operation, invalidates its copy, and moves to the I state.

6.2.3 Predictable MSI Protocol (PMSI)

For PMSI protocol, we propose two additional transient states that are necessary to guarantee that invariants are upheld. States MI^{wb} and MS^{wb} manage the write-back operation for lines in the M state. These transient states convey that: (1) a line has a pending write-back response (wb), and (2) the final state the line transitions to after the core completes the write-back. A core that has a cache line in M moves to MI^{wb} or MS^{wb} on observing a remote write or read request to the same cache line respectively. In the core's write-back allocated slot, the core completes the write-back and transitions to the I or S state respectively.

6.2.4 Predictable MESI Protocol (PMESI)

The PMESI protocol adds the exclusive (E) state to the PMSI protocol. Table 1 shows the transition to E state. Adding the E state introduces two new transient states: EI^{wb} and ES^{wb} states. These transient states manage the write-back operations for lines in the E state. Similar to the MI^{wb} and MS^{wb} states in PMSI, these states convey that a line has a pending write-back, and the final state of the line after the write-back is completed. The rationale behind these states is that when the shared memory sends exclusive data for a requested line to a core, the coherence state of this line in the shared memory is recorded as M. This enables the silent stores optimization in MESI/PMESI. Hence, the shared memory cannot send data to subsequent requests to this line until the core that has the line in E state performs a write-back of the line. As a result, a core that has a line in E state, and observes remote activity on the line must mark the line for write-back.

6.2.5 Optimized PMESI (Opt-PMESI)

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

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

accordingly changes the state of the line, and responds to pending memory requests to the same line. As a result, states EI^{wb} and ES^{wb} are no longer needed.

7 Latency Analysis

We derive the upper bound per-request latency that a core suffers when it attempts to access the shared memory. The considered system deploys one of the four predictable protocols: (1) PMSI, (2) PMESI, and (3) Opt-PMESI. Shared memory accesses from multiple cores are handled by a TDM bus arbitration scheme. We partition this latency into four components and compute the WC value of each of them. Definitions 1, 2, 3, 4, and 5 formally define these latency components. We use c_i as the core under analysis, and denote a request generated by c_i as req_i.

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

Definition 2. Access latency *is the time required to transfer the requested data by* c_i *between the shared memory and the private cache of* c_i*. We assume that this data transfer takes a fixed latency,* L^{acc}*. This latency can be considered as the WC access latency of the shared memory.*

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

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

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

Lemma 1. *The WC arbitration latency,* WCL_i^{arb}*, of a request by* c_i *is calculated by Equation (1).*

$$WCL_i^{arb} = N \cdot S. \tag{1}$$

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

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

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

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

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

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

We refer the readers to [17] for an illustrative example of the WC inter-core coherence scenario for a 3-core system. □

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

Proof. In PMESI, lines in E or M state must be written back to shared memory on observing remote memory activity on the same line (Table 1). Hence, two worst-case scenarios exists for PMESI that result in the worst-case inter-core coherence latency. The first worst-case scenario is similar to PMSI (Lemma 2) where the remaining $N-1$ cores broadcast store requests to the same line before c_i's request. The second worst-case scenario occurs when core c_j in the remaining $N-1$ cores first broadcasts a load request, and then the remaining $N-2$ cores broadcast store requests to A before c_i's request to A. In this scenario, core c_j receives A in E state as no other core has A in their private caches. Recall from Section 3 that only one core can have a line in E state. On observing remote store requests from other cores, c_j marks A for write-back, and completes the write-back in the next allocated slot. Since the remaining $N-2$ cores perform store operations, each of the $N-2$ cores must first complete the store operation,

and then write-back A. We omit a detailed proof regarding this scenario as it is similar to the proof of Lemma 2. □

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

Proof. Recall from Section 6.2.5, cores do not perform write-back responses for lines in E state in Opt-PMESI. As a result, only lines in M state trigger write-back responses in Opt-PMESI on remote memory activity and cache line replacements. Hence, the worst-case scenario for Opt-PMESI is equivalent to PMSI, and the proof is similar to that of Lemma 2. □

Lemma 5. *For PMSI, PMESI, and Opt-PMESI, $WCL_i^{interCoh}$ is calculated by Equation (2).*

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

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

Lemma 6. *For PMSI, PMESI, and Opt-PMESI, the WC intra-core coherence latency is calculated by Equation (3).*

$$WCL_i^{intraCoh} = \begin{cases} 2 \cdot N \cdot S & N > 2 \\ N \cdot S & N \leq 2 \end{cases}. \tag{3}$$

Proof. There exist two cases:

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

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

We refer the readers to [17] for an illustrative example of the WC intra-core coherence scenario for a 3-core system. □

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

$$WCL_i^{tot} = WCL_i^{arb} + WCL_i^{interCoh} + WCL_i^{intraCoh} + L^{acc}. \tag{4}$$

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

Computing Total WCET. Compositional timing analysis [27] can use the derived WCL of a memory request under predictable cache coherence protocols to compute the total WCET of an real-time task. For a real-time task that makes k memory requests, Equation (5) computes the total worst-case memory latency of a task under the predictable cache coherence protocols ($WCL_{total}^{coherence}$), and Equation (6) computes the total worst-case memory latency of a task under alternative mechanisms for predictable shared data accesses that do not use predictable cache coherence such as cache bypassing and task scheduling ($WCL_{total}^{alternative}$). For the alternative data communications, the shared data are either disallowed to be cached in the core's private caches or constrained such that shared data are cached in only one core's cache at any time instance. As a result, the WCL of a memory request under these alternative mechanisms is the sum of the arbitration latency and L^{acc}.

$$WCL_{total}^{coherence} = k \cdot (2 \cdot N \cdot S \cdot (N+1) + L^{acc}) \tag{5}$$

$$WCL_{total}^{alternative} = k \cdot (N \cdot S + L^{acc}). \tag{6}$$

It is clear from the above equations that $WCL_{total}^{coherence}$ is higher than $WCL_{total}^{alternative}$. For a 4-core multi-core platform and a real-time application where all memory requests are to shared data, $WCL_{total}^{coherence}$ is 10.25× higher than $WCL_{total}^{alternative}$. However, our empirical evaluation shows that even when all requests are to shared data, the total execution time is lower with predictable cache coherence compared to alternative mechanisms. This is because not all memory requests experience the worst-case scenario under predictable cache coherence.

The $WCL_{total}^{coherence}$ can be made tighter by using the following application information: (1) classification of memory addresses that are either shared between cores or private to cores, and (2) read/write access patterns on shared data. Data identified as private to cores and shared data that are only subjected to read requests cannot exhibit the worst-case scenarios described in Lemmas 2, 3, and 4. Rather, the worst-case scenario for a core's access to the above identified data is when it is not present in the core's private cache,

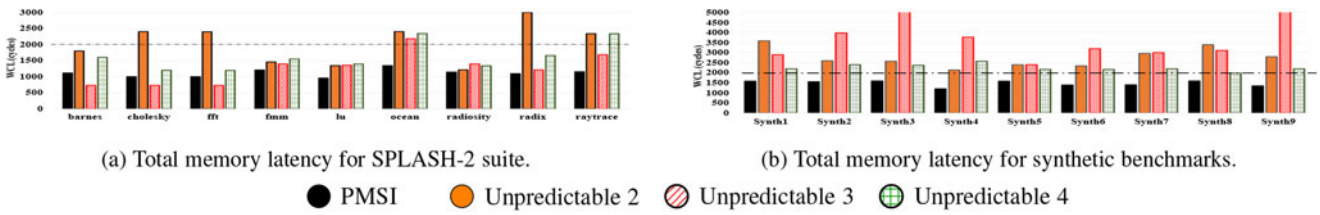


Fig. 4. Total observed WC latencies. Unpredictable i corresponds to source i in Section 5. Analytical bound highlighted.

and must be fetched from the shared memory. Hence, the worst-case memory latency for such data accesses is $N \cdot S + L^{acc}$. Equation (7) computes $WCL_{total}^{coherence}$ with this application information where $j \leq k$ is the number of memory requests that are either private to a core or marked as read only throughout the application execution.

$$WCL_{total}^{coherence} = j \cdot (N \cdot S + L^{acc}) + (k - j) \cdot (2 \cdot N \cdot S \cdot (N + 1) + L^{acc}). \tag{7}$$

Coverage of Unpredictability Sources. Section 5 listed five unpredictability sources and their associated design invariants. Missing an unpredictability source means that the WCL bound is larger than the one the above analysis provides. We argue that we covered all possible unpredictability sources.

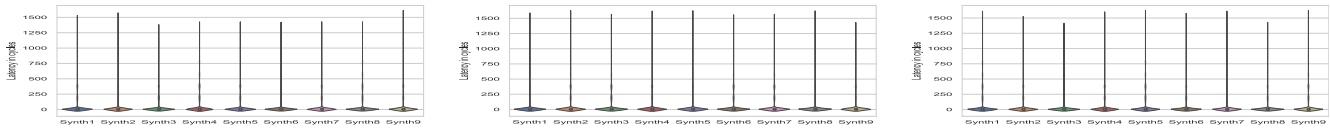
Assume that there exists an unaccounted unpredictable source. The worst-case scenario consists of memory requests across cores to data that result in the WCL. To construct the worst-case scenario, we categorize a core's (c_i) memory request into three categories: (1) the memory request is a *cache hit*, (2) the memory request is *not* a cache hit and the requested data is *neither* simultaneously cached in other cores' caches nor simultaneously requested by other cores, and (3) the memory request is *not* a cache hit and the requested data is *either* simultaneously cached in at least another core's cache or simultaneously requested by at least another core. Memory requests in (1) do not access the bus to broadcast coherence messages or wait for the requested data. Hence, the worst-case scenario cannot consist of memory requests in (1). Memory requests in (2) access the bus to broadcast coherence messages and wait for the requested data. However, since other cores neither simultaneously cache nor make requests to the same data, $WCL_i^{interCoh}$ is 0. Hence, the worst-case scenario cannot consist of memory requests in (2). Memory requests in (3) also access the bus to broadcast coherence messages and wait for the requested data. Furthermore, the requested data is either simultaneously cached in another cores' caches or simultaneously requested by other cores. Hence, in the worst-case, c_i must wait for other cores' to complete their requests and perform any actions (write-backs) before receiving the requested data. Therefore, WCL_i^{arb}, $WCL_i^{interCoh}$, and $WCL_i^{intraCoh}$ are $\neq 0$, and the worst-case scenario must consist of memory requests in (3). However, the scenarios described in the analysis in Section 7 do indeed consist of memory requests that fall in (3). Furthermore, these scenarios are the worst-case scenarios that result in the WCL. Since the latency analysis are for protocols that satisfy the design invariants listed in Section 5, we have covered all possible unpredictable scenarios.

8 Evaluation

We integrate PMSI, PMESI, and Opt-PMESI into the gem5 simulator [18]. We use the Ruby memory model in gem5, which is a cycle-accurate model with a detailed implementation of cache coherence events. We use a multi-core architecture that consists of in-order x86 cores running at 2GHz. Each core has a private 16KB direct-mapped L1 cache, with its access latency as 3 cycles. All cores share an 8-way set-associative 1MB LLC cache. Since the focus of this work is on coherence interference, we use a perfect LLC cache to avoid extra delays from accessing off-chip DRAM. Consequently, the access latency to the LLC is fixed, and equals to 50 cycles ($L^{acc} = 50$ cycles). The DRAM access overheads can be computed using other approaches such as [22], [23], and they are additive [28] to the latencies derived in this work. Both L1 and LLC have a cache line size of 64 bytes. The interconnect bus uses TDM arbitration amongst cores. The L1 cache controller uses work-conserving TDM arbitration between a core's own requests and its responses to other core requests. We do not run an operating system in the simulator, and hence, all memory addresses generated by the cores are physical memory addresses. We evaluate PMSI, PMESI, and Opt-PMESI using the *SPLASH-2* [29] benchmark suite. In addition, we use synthetic workloads to stress the WC behavior. We used the verification process described in [17] to verify PMESI and Opt-PMESI protocols. We also formally verified the correctness properties and WCL bounds for the protocols using the formal models developed in [30].

8.1 Observed Worst-Case Latencies

We study the effectiveness of PMSI, PMESI, and Opt-PMESI to bound the delays resulting from coherence interference. We also study the effects of violating each one of the invariants on the memory latency. We use a 4-core system for our experiments. For SPLASH-2, we launch each SPLASH-2 application as four threads using four single-threaded cores, where only one application is used per experiment. Fig. 4 depicts our findings, and shows the total observed memory latency. Since SPLASH-2 applications are optimized to minimize data sharing, they do not stress the coherence protocol. Therefore, to further stress the coherence protocol, we execute synthetic experiments using 9 synthetically-generated workloads: Synth1 to Synth9 in Fig. 4. In each synthetic experiment, we simultaneously run four identical instances of one workload by assigning one instance on each core. These experiments represent the maximum possible sharing of data since each core generates the same sequence of memory requests. The WC arbitration latency for benchmarks in all experiments is $N \cdot S = 200$ cycles for $N = 4$



(a) PMSI on synthetic workloads. (b) PMESI on synthetic workloads. (c) Opt-PMESI on synthetic workloads.

Fig. 5. Memory request latency distribution under predictable cache coherence protocols for synthetic workloads.

cores and slot $S = L^{acc} = 50$ cycles; hence, not shown. Since all three protocols have the same worst-case scenarios and latencies (Section 7), we present results only for the PMSI protocol. We verified that the below observations also apply to PMESI and Opt-PMESI. Fig. 5 shows the violin plot distributions of memory request latency under different predictable cache coherence protocols.

Observations. (1) Fig. 4 shows that for PMSI the total WC latencies are within their analytical total WCL bounds. We also observed that the individual latency components such as the arbitration, inter-core, and intra-core coherence latency components are within their respective analytical WCL bounds derived in Section 7, and refer the readers to [17] for these results. (2) On the other hand, violating any of the invariants introduces a source of unpredictability, which results in exceeding those bounds. Moreover, for source 1, one of the cores is not able to obtain an access to a block that it requests and the program never terminates. This is the reason that Fig. 4 does not show Unpredictable 1. This shows that augmenting a conventional coherence protocol with a predictable arbiter does not guarantee predictability. Note that violating some of the invariants also results in exceeding the latency bounds of the individual latency components. For example, we observed that violating invariant 3 causes resulted in the observed inter-core coherence latency to exceed the corresponding analytical bound across all synthetic and SPLASH-2 benchmarks. We refer the readers to [17] that shows the impact of violating the invariants on the individual latency components. (3) For a quad-core system, the latency suffered by a core due to coherence interference is $9\times$ more than the latency due to bus arbitration. The inter-core coherence interference solely contributes a latency up to $7\times$ of the arbitration latency, while the latency resulting from the intra-core coherence interference is double the arbitration latency. This provides evidence of the importance of considering the coherence latency when sharing data across multiple cores for real-time applications. (4) From the violin plot distributions in Fig. 5, most memory requests to shared data under PMSI, PMESI, and Opt-PMESI protocols benefit from caching, and experience lower memory request latency. This highlights the key benefit of using predictable cache coherence protocols compared to alternative predictable shared data mechanisms that constrain private caching of shared data. The maximum observed memory request latency across synthetic benchmarks under PMSI, PMESI, and Opt-PMESI are within the analytical WCL bound.

8.2 Comparison Against Prior Predictable Approaches

We compare the overhead caused by four alternative predictable approaches to handle data sharing in multi-core real-time systems: (1) not using private caches (*uncache-all*), (2) not caching the shared data (*uncache-shared*), (3) the proposed PMSI, PMESI, and Opt-PMESI protocols, and (4) mapping all tasks that share data to the same core (*single-core*). For the first three approaches, each application is distributed across four-cores. *uncache-shared* is an adaptation of the approach by [7], [9], but for data instead of instructions. *single-core* maps tasks with shared data to the same core to eliminate incoherence due to shared data, which adopts the idea of data-aware scheduling [14]. The overhead is calculated as the slowdown compared to the conventional MESI protocol. Fig. 6 depicts our findings for the SPLASH-2 workloads. We focus more on the performance gains of PMESI and Opt-PMESI, and refer the reader to [17] for a detailed comparison.

Observations. (1) Across all benchmarks, the *uncache-all* has the worst execution time with a geometric mean slowdown of $32.66\times$ compared to MESI, followed by *single-core* and *uncache-shared* with geometric mean slowdowns of $2.67\times$ and $2.11\times$ respectively. The *uncache-shared* and *single-core* approaches require additional hardware and software modifications to the applications and RTOS to track cache lines shared between cores. (2) On the other hand, PMSI, PMESI, and Opt-PMESI protocols achieve better performance compared to all other predictable approaches with no changes to the application or RTOS. PMSI, PMESI, and Opt-PMESI achieve improved performance of up to $4\times$ the best competitive approach, *uncache-shared*, with a geometric mean slowdown of $1.46\times$, $1.59\times$, and $1.42\times$ in performance compared to MESI respectively. (3) For the synthetic benchmarks, the *single-core* approach offers $2.9\times$ average performance speedup over *uncache-shared* approach. This is because the *uncache-shared* approach disallows any caching of memory addresses, and hence, no memory requests result in cache hits. The PMSI, PMESI, and Opt-PMESI protocols offer $3.08\times$, $2.99\times$, and $3.12\times$ average performance speedup over *uncache-shared* respectively. Compared to *single-core*, PMSI, PMESI, and Opt-PMESI exhibit performance speedups as high as 16 percent without constraining core utilization.

8.3 Comparison of PMSI, PMESI, and Opt-PMESI

Figures 7a and 7b compare the average-case performance of the PMSI, PMESI, and Opt-PMESI protocols for the synthetic and SPLASH-2 benchmarks respectively against

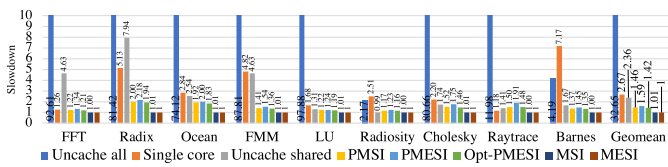
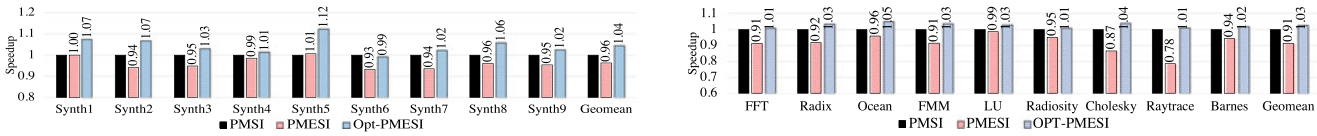


Fig. 6. Execution time slowdown compared to MESI protocol.



(a) Synthetic benchmarks.

(b) SPLASH-2 benchmark suite.

Fig. 7. Average-case performance speedups of PMSI, PMESI, and Opt-PMESI for synthetic and SPLASH-2 benchmarks.

PMSI. While all the protocols have the same WCL bounds, PMESI and Opt-PMESI have additional states and transitions that enable average-case performance improvements over PMSI.

Observations. (1) Across synthetic and SPLASH-2 benchmarks, PMESI does not provide performance benefits over PMSI. The key reason for this is the increased number of write-backs in PMESI due to states E and M. In PMESI, a core triggers a write-back for a line that it has in E or M state. Recall from Section 6.1 that cores deploy a predictable arbitration scheme that services write-backs and pending demand requests in a core's allocated slot. As a result, the increased number of write-backs in PMESI contend for the allocated slots resulting in longer execution time to complete cores' demand requests. For the synthetic benchmarks, we observed that PMESI experiences 24 percent more write-backs on average than PMSI. For the SPLASH-2 benchmarks, PMESI experiences 1.9× more write-backs than PMSI. This is because the working data set sizes of the SPLASH-2 benchmarks do not fit in the private caches resulting in more cache line evictions due to capacity misses. As a result, in PMESI, 44 percent of the total write-backs in SPLASH-2 are due to cache line evictions to lines in E state. Hence, PMESI does not improve over PMSI (4 percent average performance degradation for synthetic benchmarks and 9 percent average performance degradation for SPLASH-2 benchmarks) due to the increased number of write-back responses that contend for allocated slots with demand requests. (2) The additional hardware overhead in Opt-PMESI addresses this performance limitation of PMESI, and improves over PMSI and PMESI for both the synthetic and SPLASH-2 benchmarks. For synthetic and SPLASH-2 benchmarks, Opt-PMESI improves performance by 4 and 3 percent respectively. The performance improvement is primarily due to silent stores that allows cores to complete stores on lines in E state without broadcasting on the bus.

9 Conclusion

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

References

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, pp. 46–61, 1973.

[2] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware support for WCET analysis of hard real-time multicore systems," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 57–68.

[3] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *Proc. 9th Eur. Dependable Comput. Conf.*, 2012, pp. 132–143.

[4] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst, "System level performance analysis for real-time automotive multicore and network architectures," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 28, no. 7, pp. 979–992, Jul. 2009.

[5] M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, and F. D. Smith, "Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems," in *Proc. IEEE Real-Time Syst. Symp.*, 2016, pp. 57–68.

[6] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst, "Communication centric design in complex automotive embedded systems," in *Proc. 29th Euromicro Conf. Real-Time Syst.*, 2017, pp. 10:1–10:20.

[7] D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches," in *Proc. 30th IEEE Real-Time Syst. Symp.*, 2009, pp. 68–77.

[8] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *Proc. 25th Euromicro Conf. Real-Time Syst.*, 2013, pp. 157–167.

[9] B. Lesage, D. Hardy, and I. Puaut, "Shared data caches conflicts reduction for WCET computation in multi-core architectures," in *Proc. 18th Int. Conf. Real-Time Netw. Syst.*, 2010, Art. no. 2283.

[10] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Commun. ACM*, vol. 55, pp. 78–89, 2012.

[11] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," in *Synthesis Lectures on Computer Architecture*, San Rafael, CA, USA: Morgan & Claypool, 2011.

[12] G. Gracioli and A. A. Fröhlich, "On the design and evaluation of a real-time operating system for cache-coherent multicore architectures," *ACM SIGOPS Operating Syst. Rev.*, vol. 49, pp. 2–16, 2016.

[13] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, 2015, Art. no. 36.

[14] J. M. Calandrino and J. H. Anderson, "On the design and implementation of a cache-aware multicore real-time scheduler," in *Proc. 21st Euromicro Conf. Real-Time Syst.*, 2009, pp. 194–204.

[15] G. Gracioli and A. A. Fröhlich, "Two-phase colour-aware multicore real-time scheduler," *IET Comput. Digit. Techn.*, vol. 11, no. 4, pp. 133–139, Jul. 2017.

[16] 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 *Proc. Int. Conf. Embedded Comput. Syst. Architectures Model. Simul.*, 2014, pp. 107–114.

[17] M. Hassan, A. M. Kaushik, and H. Patel, "Predictable cache coherence for multi-core real-time systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2017, pp. 235–246.
[18] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, 2011.
[19] V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *Proc. 45th ACM/IEEE Des. Autom. Conf.*, 2008, pp. 300–303.
[20] M. Schoeberl, W. Puffitsch, and B. Huber, "Towards time-predictable data caches for chip-multiprocessors," in *Proc. IFIP Int. Workshop Softw. Technol. Embedded Ubiquitous Syst.*, 2009, pp. 180–191.
[21] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *Proc. 9th IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codesign Syst. Synthesis*, 2011.
[22] Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst case analysis of DRAM latency in multi-requestor systems," in *Proc. IEEE 34th Real-Time Syst. Symp.*, 2013, pp. 372–383.
[23] M. Hassan, H. Patel, and R. Pellizzoni, "A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2015, pp. 307–316.
[24] B. Lesage, I. Puaut, and A. Seznec, "PRETI: Partitioned real-time shared cache for mixed-criticality real-time systems," in *Proc. 20th Int. Conf. Real-Time Netw. Syst.*, 2012, pp. 171–180.
[25] J. Bin, S. Girbal, D. G. Perez, A. Grasset, and A. Merigot, "Studying co-running avionic real-time applications on multi-core COTS architectures," in *Proc. Embedded Real Time Softw. Syst. Conf.*, 2014, pp. 1–10.
[26] Cortex, ARM, "A9 MPCore Technical Reference Manual," *June Rev r4p1*, 2012.
[27] S. Hahn, J. Reineke, and R. Wilhelm, "Towards compositionality in execution time analysis: Definition and challenges," *ACM SIGBED Rev.*, vol. 12, pp. 28–36, 2015.
[28] H. Yun, R. Pellizzon, and P. K. Valsan, "Parallelism-aware memory interference delay analysis for COTS multicore systems," in *Proc. 27th Euromicro Conf. Real-Time Syst.*, 2015, pp. 184–195.
[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd Annu. Int. Symp. Comput. Architecture*, 1995, pp. 24–36.
[30] N. Sensfelder, J. Brunel, and C. Pagetti, "Modeling Cache Coherence to Expose Interference," in *Proc. 31st Euromicro Conf. Real-Time Syst.*, 2019, pp. 18:1–18:22.

Anirudh Mohan Kaushik (Member, IEEE) is currently working toward the PhD degree with the Electrical and Computer Engineering Department, University of Waterloo, Waterloo, ON, Canada. His research interests include real-time embedded systems architecture and high performance computer architecture.

Mohamed Hassan (Member, IEEE) received the MSc degree from Cairo University, Giza, Egypt, in 2012, and the PhD degree from the University of Waterloo, Waterloo, ON, Canada, in 2017. He is an assistant professor at McMaster University, Hamilton, ON, Canada. His current research interests include real-time embedded systems, computer architecture, hardware validation, and security.

Hiren Patel (Member, IEEE) is a professor with the Electrical and Computer Engineering Department, University of Waterloo, Waterloo, ON, Canada. His current research interests include real-time embedded systems, computer architecture, and system level design methodologies.