PREPRINT: "M. Hassan, A. Kaushik, and H. 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, pp. 1-12."

Predictable Cache Coherence for Multi-Core Real-Time Systems

Mohamed Hassan, Anirudh M. Kaushik and Hiren Patel
{mohamed.hassan, anirudh.m.kaushik, hiren.patel}@uwaterloo.ca
University of Waterloo, Waterloo, Canada

***Abstract*—This work addresses the challenge of allowing simultaneous and predictable accesses to shared data on multi-core systems. We propose a predictable cache coherence protocol, which mandates the use of certain invariants to ensure predictability. In particular, we enforce these invariants by augmenting the classic modify-share-invalid (MSI) protocol with transient coherence states, and minimal architectural changes. This allows us to derive worst-case latency bounds on predictable MSI (PMSI) protocol. Our analysis shows that while the arbitration latency scales linearly, the coherence latency scales quadratically with the number of cores, which emphasizes that importance of accounting for cache coherence effects on latency bounds. We implement PMSI in gem5, and execute SPLASH-2 and synthetic workloads. Results show that our approach is always within the analytical worst-case latency bounds, and that PMSI improves average-case performance by up to 4× over the next best predictable alternative. PMSI has average slowdowns of 1.45× and 1.46× compared to MSI and MESI protocols, respectively.**

I. Introduction

In hard real-time systems, correctness depends not only on the functioning behavior, but also 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 that application has to be less than or equal to this upper bound to achieve predictability. As application demands continue to increase from the avionics [3] and automotive [4] domains, there is a surge in wanting to use multi-core platforms for their deployments. This is primarily due to the benefits multi-core platforms provide in cost, and performance. However, multi-core platforms pose new challenges towards guaranteeing temporal requirements of running applications.

One such challenge is in maintaining coherence of shared data stored in private cache hierarchies of multi-cores known as *cache coherence*. Cache coherence is realized by implementing a protocol that specifies a core's activity (read or write) on cached shared data based on the activity of other cores on the same shared data. While cache coherence can be implemented in software or hardware, modern multi-core platforms implement the cache coherence protocol in hardware. This is so that software programmers do not have to explicitly manage coherence of shared data in the application. A recent work studied the effect of cache coherence on execution time using different Intel and AMD processors and coherence protocols [5]. 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.

978-1-5090-5269-1/17/$31.00

This emphasizes the importance of considering cache coherence effects when deriving WCET bounds. However, as observed by a recent survey [6], there is no existing technique to account for the effects of coherence protocols in static timing analysis in real-time systems. As a result, tasks on multi-core systems cannot coherently and predictably share data unless some restrictions to eliminate those effects are enforced. For instance, a possible solution is disabling the caching of shared data [7], [8]. Clearly, this solution significantly increases the execution time of applications with shared data, which may render applications unschedulable. Another solution prohibits tasks with shared data from running simultaneously on different cores using task scheduling [9]. However, this solution requires special hardware performance counters and modifications to currently available scheduling techniques. A third solution suggests modifying the applications by marking instructions with shared data as critical sections such that they are accessed by only a single core at any time instance [10]. Although this allows caching of shared data, it stalls all tasks but one from accessing the data, which in worst case (WC) amounts to sequentially running the tasks.

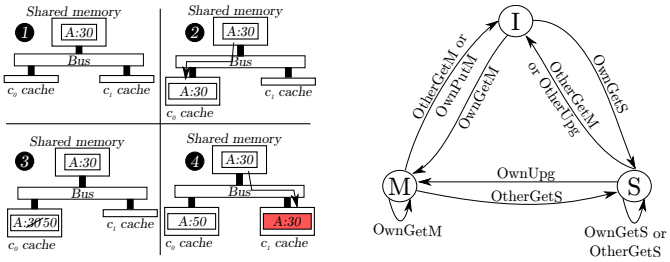
In summary, existing solutions prohibit tasks from simultaneously accessing shared data. This approach successfully avoids data incoherence, but it does so at the expense of one or more of the following: 1) severely degrading performance, 2) imposing scheduling restrictions, 3) imposing source-code modifications, and 4) requiring hardware extensions. This work presents a predictable cache coherence protocol to allow for simultaneous accesses to shared data. The proposed solution provides considerable performance improvements, does not impose any scheduling restrictions, and does not require any source-code modifications. We address the problem of maintaining cache coherence in multi-core real-time systems by modifying a current coherence protocol such that data sharing is viable for real-time systems in a manner amenable for timing analysis. Doing so, we make the following contributions:

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

2) We show the unpredictable behaviors and the proposed solutions to rectify them using the Modified-Shared-Invalidate (MSI) coherence protocol. Accordingly, we propose predictable MSI (PMSI), a coherence protocol that fulfills the proposed invariants on MSI by introducing a set of transient states while making minimal architectural changes to cache controllers (Section VI). We release the implementation of PMSI [11] for researchers to use and extend. Modern commodity multi-core architectures implement various protocols that are optimizations of MSI such as MESI, MOESI and MESIF [12], [13]. Accordingly, our observations and proposed invariants are applicable to modern cache coherence protocols with those optimizations.
3) We provide a timing analysis for our proposed coherence protocol and decompose the analysis to highlight the contributions to latency due to arbitration logic and communication of coherence messages between cores (Section VII).
4) We evaluate the proposed coherence protocol using the gem5 simulator [14] (Section VIII). Our evaluation shows that cache coherence can increase the memory latency up to $10\times$ in a quad-core system. This further emphasizes the importance of providing safe bounds that account for the effect of cache coherence. Performance evaluation shows that PMSI achieves up to $4\times$ speedup over competitive approaches.

II. Related Work

Recent research efforts investigated the access latency overhead resulting from shared buses [2], [15], caches [16]–[18], and dynamic random access memories (DRAMs) [19]–[21]. 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 [16] or locking mechanisms [17]. Authors in [18] promote splitting the data cache into multiple data areas (e.g. stack, heap,..etc.) to simplify the analysis. However, they indicate that the coherence is still an issue that has to be addressed. Similarly, several proposals for shared main memories deployed data isolation by assigning a private memory bank per core [19], [20]. However, we find that data isolation suffers from three limitations. 1) They disallow sharing of data between tasks; thus, disabling any communication across applications or threads of parallel tasks running on different cores. 2) It may result in poor memory or cache utilization. For instance, a task may keep evicting its cache lines if it reaches the maximum of its partition size, while other partitions may remain underutilized. 3) It does not scale with increasing number of cores. For example, the number of cores in the system has to be less than or equal to the number of DRAM banks to be able to achieve isolation at DRAM. Recent works [21]–[23] recognized these limitations, and offered solutions for sharing data. Authors in [21] share the whole memory space between tasks for main memory, and [22], [23] suggested a compromise by dividing the memory space into private and shared segments for caches. Nonetheless, these approaches focus on the impact of sharing memory on timing analysis, and they do not address the problem of data correctness resulting from sharing memory. Authors of [24] study the overhead effects of co-running applications on the timing behavior in the avionics domain, where coherence is one of the overhead sources. A recent survey [6] 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 [9] proposed data sharing-aware scheduling policies that avoid running tasks with shared data simultaneously. A similar approach proposed by [5] 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 [9] is not adequate for partitioned scheduling mechanisms. A different solution introduced in [10] applied source-code modifications to mark instructions with shared data as critical sections. These critical sections were protected by locking mechanisms such that they were accessed only by a single core at any time instance. This solution suffers from certain limitations. 1) It exposes cache coherence to the software to assure correctness of shared data. 2) Only one core can access a cache line of shared data at a time. Other cores requesting this data have to stall. In worst case, this is equivalent to sequential execution. 3) Additionally, they still require hardware to keep track of whether each cache line is shared or not. PMSI allows tasks to simultaneously access shared data, which considerably improves performance. In addition, it does not pose any requirements on task scheduling techniques, and it does not require software modifications. We also provide a timing analysis that accounts for memory coherence for PMSI.



(a) Incoherent sharing of data. (b) MSI state diagram.

Fig. 1: Cache coherence.

III. 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. Figure 1a shows one instance of data incoherence on the shared cache line A in a dual-core system. ❶ Initially, the shared memory has A with a value of 30. ❷ Core c_0 performs a read on A; hence, it obtains a local copy of A in its private cache. ❸ Afterwards, c_0 executes a write operation that updates this local copy to 50. ❹ When c_1 reads A, the shared memory responds with the old value of A, 30. This is because c_0 did not update the shared memory with the new value of A; thus, c_1 obtains a stale (incorrect) version of A.

A coherence protocol avoids data incoherence by deploying a set of rules to ensure that cores access the correct version of data at all times. Usually, the coherence protocol maintains

data coherence at cache line granularity, which is a fixed size collection of data. A state machine implements these rules with a set of states representing read and write permissions on the cache line, and transitions between states denoting the activity of all cores on the shared data. The cache controller typically implements the coherence protocol. General purpose processors deploy different variants of coherence protocols. Most of them consist of three fundamental stable states, which establish the MSI protocol: *modified* (M), *shared* (S), and *invalid* (I) [25]. Figure 1b presents these states and the transitions amongst them. 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. A cache line in ***shared state*** means that the core has a valid, yet unmodified version of that line. One or more cores can have versions of the same cache line in shared state to allow for fast read accesses. A cache line in ***invalid state*** denotes the unavailability of that line in the cache or that its data is outdated. A cache controller changes the state of the cache line by observing the bus for coherence messages related to the same cache line by other cores, known as *bus snooping cache coherence* or receiving action messages from a centralized shared cache controller, known as *directory-based cache coherence*. In this 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* are messages that represent data sent or received by a core as a consequence of a core's activity. If a core requests a cache line A for reading, it issues a GetS(A) message. If it requests A for writing, it issues a GetM(A) message. If A is modified, before replacing A, the core has to issue a PutM(A) message to write it back to the shared memory. If the core has A in a shared state and wants to modify it, it issues an Upg(A) message. A core observes its own messages on the bus as well as messages by other cores. We refer to the former as Own, and to the latter as Other. For instance, in Figure 1b, when the core has a line in S state and observes its OwnUpg(), it moves to M state. In contrast, if it observes a GetM() issued by another core, OtherGetM(), it changes its state to I.

To fix the data incoherent scenario, we apply the MSI protocol to the example in Figure 1a; c_0 first issues a GetS(A) to obtain A for reading, and then issues an Upg(A) to modify it. When c_1 issues a GetS(A) to obtain A for reading, c_0 observes an OtherGetS(A) on the bus. Hence, it either sends the updated data to c_1 directly, and writes back A to shared memory, or it writes back A to the shared memory and c_1 reads the new data from it. The former case is possible only if the architecture allows for direct cache-to-cache communication. In both cases, c_1 reads the updated data and c_0 moves to the shared state. The cache coherence protocol is responsible for orchestrating such communication and transfer of data.

A. Transient Cache Coherence States

Interconnecting cores with an atomic in-order bus prevents all other cores from utilizing the bus until the core that was granted access to the bus completes its request. Consequently, most modern systems implement non-atomic reordering buses for improved performance, where the stall time of one core waiting for data response can be used to service requests of other cores. One example is the QuickPath interconnect [27] from Intel that is used for inter-processor communication. However, if the bus is not an atomic in-order one, then a memory request to a cache line may be intervened by other requests to the same cache line before it completes (non-atomicity), and coherence messages can be reordered by the bus (reordering bus). Hence, a set of transient states between stable states is required to capture events caused by intervening coherence messages due to a non-atomic bus architecture. We categorize the transient states imposed by a non-atomic reordering bus into two categories based on their semantics.

1) **Transient states for coherence messages.** These transient states denote that a core is waiting for its own coherence message to be placed on the bus. A core's own coherence message may be delayed on the bus due to the presence of other messages on the bus. For instance, when a core c_i has a read request to an invalid line, it issues an OwnGetS() 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 [25]. In this case, c_i moves to a transient state IS^a waiting for its OwnGetS() to appear on the bus before moving to the stable state, S. IS^a is between the two stable states, I and S. Similar transient states exist between other stable states [25].
2) **Transient states for data messages.** These states denote that a core is waiting for data either from a core that has the data in its private cache hierarchy or from the shared memory. For example, IS^d transient state denotes that a core issued a GetS() and did not receive a data response yet. IS^d is between the two stable states, I and S. Again, similar transient states exist between other stable states.

IV. 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 can have shared data. These tasks can belong to a parallel application that is distributed across cores, or different applications that communicate between each other. Cores can share the whole shared memory space similar to [21] or share part of the memory space similar to [23]. We do not impose any restrictions on how the interference on the shared memory is resolved, whether it is the LLC or the DRAM. Furthermore, we do not require any special demands from the task scheduling mechanism. This allows one to integrate the proposed solution to current task scheduling techniques, and to various mechanisms that control accesses to shared memories in real-time systems. Cores share a common bus connecting private caches to the shared memory, where data transfers amongst private caches are only via the shared memory (no cache-to-cache transfer). Although some of the problems addressed in this paper may also apply to systems supporting cache-to-cache transfer, those systems are not the focus of this paper. The common bus transfers data and coherence messages between the shared memory and the private

caches. For example, Figure 1a shows a two core setup where the cores are connected to each other and to the shared memory via a common bus. The common bus also transfers coherence messages deployed by the coherence protocol to ensure data correctness. The system deploys a predictable arbitration on the common bus. The proposed solution is independent of the core architecture, and the arbitration mechanism on the bus. However, the analysis and experiments we present in this work consider a system with in-order cores, and a time-division-multiplexing (TDM) bus as the base arbitration scheme to derive WC latencies. TDM can be either work-conserving or non work-conserving. Work-conserving TDM grants the slot to the next core if the current core does not have pending requests, while in non work-conserving TDM such slot remains idle [21]. We use a TDM slot width that allows for one data transfer between shared memory and the private cache including the overhead of necessary coherence messages.

V. Sources of Unpredictability Due to Coherence

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

Investigating the implications of a conventional coherence protocol on the WCET, we find that there are four major sources that can lead to unpredictable behavior. Figure 2 illustrates example scenarios for these sources. For each source, Figure 2 delineates the example scenario on the left with the source of unpredictable behavior shaded in red. Figure 2 also illustrates how satisfying the proposed invariants prevents these actions and leads to a bounded memory latency. Figure 2 considers a system with three cores, c_0, c_1, and c_2, and deploys a TDM arbitration amongst their requests to the common bus. If the request type is not specified whether it is a read or write, that means the scenario is agnostic to it. Each of Figures 2a–2e separately defines the initial system state and the core under analysis for the corresponding scenario.

A. Source 1: Inter-core Coherence Interference on Same Line

If a core requests a line that has been modified by another core, it has to wait for the modifying core to write back that line to the shared memory. Source 1 occurs when multiple cores request the same line, say A, where A is modified by another core. In Figure 2a, initially, c_0 has a modified version of A in its private cache. ❶ c_2 issues a read request to A. Since c_0 has the modified version of A, it has to write back its data to the shared memory first. However, this is c_2's slot; thus, c_0 has to wait for its slot to perform the write back. ❷ c_0 writes back A to the shared memory in its slot. ❸ c_1 issues a write request to A. Since the shared memory has the updated version of A, c_1 is able to obtain A and modify it. ❹ c_2 reissues a read request to A. This time c_2 has to wait for c_1 to write back A. From c_2's perspective, slot ❹ is a repetition of ❶; c_2 reissues its request to A and waits for another core to write it back. Thus, this situation is repeatable and can result in unbounded memory latency. Although c_2 is granted access to the bus, it is unable to obtain the requested data due to the coherence interference.

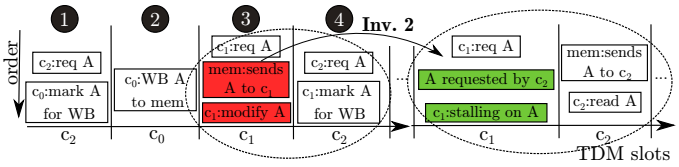
Proposed solution. We avoid this problem by enforcing Invariant 2. Invariant 2 requires memory to service requests to the same cache line in their arrival order; thus, it guarantees that a line being requested by a core will not be invalidated before the core accesses it. Imposing Invariant 2 in Figure 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.

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

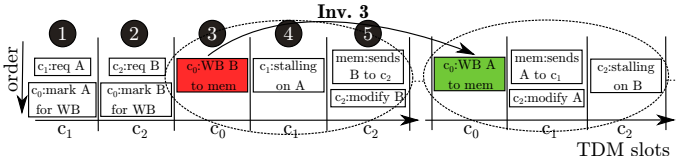
B. Source 2: Inter-core Coherence Interference on Different Lines

This source of interference arises when a core has multiple pending lines to write back because other cores requested them. This is an indirect source of interference. Cores requesting access to different lines can interfere with each other because of the coherence protocol. For instance in Figure 2b, c_0 has modified versions of lines A and B. The core under analysis is c_1. c_1 and c_2 issue requests to A ❶ and B ❷ during their corresponding slots. Accordingly, c_0 has to write back both A and B to the shared memory. Since c_0 is permitted to conduct one memory transfer in a slot, at ❸, it can write back only one line. If no predictable mechanism manages the write backs, c_0 can pick any pending one. At ❸, c_0 writes back B. Therefore, at ❹, c_1 is stalling on A. This situation can repeat indefinitely. While c_1 is waiting for A, c_2 can ask for another line, say C, which is also modified by c_0 and the same situation can repeat.

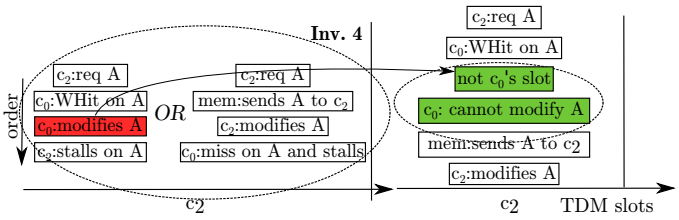
Proposed solution. We avoid this situation by enforcing Invariant 3. Invariant 3 imposes an order in servicing coherence messages from other cores (write backs, for example). The right side of Figure 2b deploys Invariant 3. Since the request



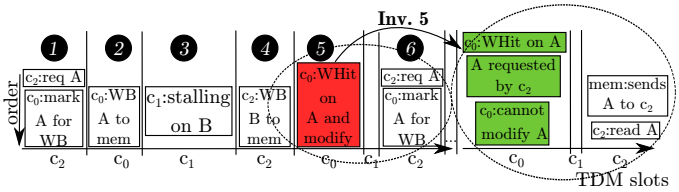
(a) Source 1: initially, c_0 modified A. c_2 is under analysis



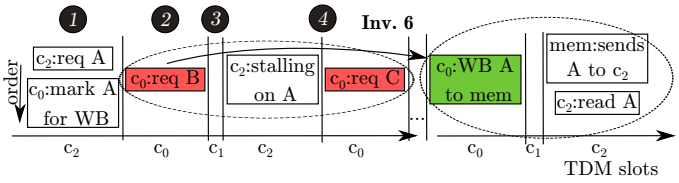
(b) Source 2: initially, c_0 modified A and B. c_1 is under analysis.



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



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



(e) Source 4: initially, c_0 has modified A. c_2 is under analysis.

Fig. 2: Scenarios of unpredictable behavior.

to A arrives before that to B, c_0 has to write back A first then B; thus, a predictable behavior is guaranteed.

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

C. Source 3: Inter-core Interference Due to Write Hits

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

1) Source 3A: Inter-core interference due to write hits to non-modified lines during another core's slot: Figure 2c exemplifies this scenario. c_0 has a version of A in its private cache that is not modified. During c_2's slot, c_2 issues a write request to A, while simultaneously c_0 has a write operation to A that results in a hit in its private cache. This creates a race with two possibilities. If c_0's write hit on A occurs first, c_2 has to wait until c_0 writes back A. On the other hand, if c_2's request appears on the bus first, c_0 has to invalidate its own local copy of A. Hence, c_0's request to A will be a miss and has to wait for c_2 to write back A before it gets another access to it. Assume that c_0's write hit occurs first and c_2 waits. After c_0 writes back A and during c_2's next slot, c_0 again has another write hit to A. Again, c_2 has to wait for c_0 to write back A. Consequently, this situation is repeatable and can starve c_2.

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

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

2) Source 3B: Inter-core interference due to write hits to non-modified lines that are requested by another core: Invariant 4 resolves the race situation between a request generated by a core in its designated slot and write hits from other cores. Note that the write hits to non-modified lines can lead to another unpredictable situation that Invariant 4 does not manage. We illustrate this situation in Figure 2d. Initially, c_0 has a modified version of A, c_2 has a modified version of B, and c_1 has requested B. ❶ c_2 requests A to read; thus, in c_0's next slot, it updates the shared memory with the modified value of A ❷. Since c_2's request is a read, c_0 does not invalidate its local version of A. At ❹, c_2 has two pending actions: fetching A from memory, and writing back B to the memory in response to c_1's request. Assume that c_2 chooses to write back B. Therefore, its request to A waits for the next slot. At ❺, c_0 has a write hit to A. Consequently, since this is c_0's slot, it conforms with Invariant 4; thereby, it modifies A. At ❻, it has to reissue its request to A and wait for c_0 to write back A to memory again. From c_2's perspective, this situation is similar to the situation at ❶. Similarly, in 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.

Proposed solution. We avoid the unbounded latency by enforcing Invariant 5. Invariant 5 stalls a write request, which is a hit to a non-modified line until all waiting requests from previous slots are completed. Thereby, it avoids the aforementioned unpredictable consequences. Maintaining Invariant 5 in the right side of Figure 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 issued afterwards in the corresponding slot.

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

D. Source 4: Intra-core Coherence Interference

This interference is between two actions from the same core. The first action is its own pending request, while the second is its response to a request from another core. This response is for example, a write back to a line that this core has in a modified state. In Figure 2e, c_0 has a modified version of A. ❶ c_2 requests A; thus, c_0 marks A to write back in its next slot. However, at ❷ c_0 is in its next slot and has its own request to B pending to issue to the bus. Thus, the write back of A waits for c_0's next slot. Similarly, at ❹, it has another pending request to another line, C. Accordingly, the write back of A by c_0 can indefinitely stall, which results in unbounded latency of c_2's request.

Proposed solution. We introduce Invariant 6 to resolve this intra-core interference problem predictably. Invariant 6 states that any predictable arbitration mechanism between coherence requests of a core and responses from the same core is sufficient to address the intra-core interference. Deciding the adequate arbitration depends on the application. Deploying Invariant 6 in Figure 2e, the predictable arbitration mechanism will eventually allocate one c_0's slot to the write back operation of A. Thus, the memory latency of c_2's request is bounded.

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

VI. PMSI: A Predictable Coherence Protocol

We show the effectiveness of the proposed invariants by applying them to the conventional MSI protocol. This results in a predictable MSI (PMSI) protocol for multi-core real-time systems. To ensure these invariants are held, we propose architectural modifications and additional coherence states. The proposed architectural modifications satisfy Invariants 1 and 2 without any changes to the coherence protocol. However, Invariants 3–6 require modifications to both the architecture and the coherence protocol. This is because Invariants 3 and 6 regulate the write back operation of cache lines. Since a core has to wait for a designated write back slot to write back a cache line A, it has to maintain A in a transient state to indicate that A is waiting for write back. Similarly, Invariants 4 and 5 regulate the write hit operation to non-modified lines. A core has to wait for a designated slot to perform the write hit operation to a cache line, say B. Accordingly, it has to maintain B in a transient state indicating that it has a pending write to B.

A. Architectural Modifications

Figure 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

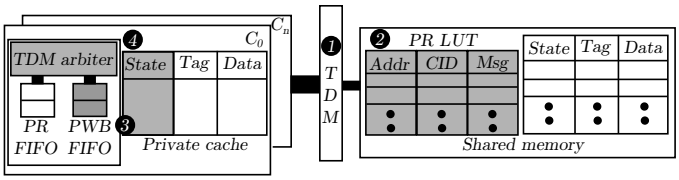


Fig. 3: Architectural changes necessary for PMSI.

memory. The proposed architecture changes are highlighted in grey. In our four-core evaluation system, the storage overhead is less than 128 bytes.

❶ The TDM arbiter manages the coherence requests such that each core can issue a coherence request message only when it is granted an access to the bus. This satisfies Invariant 1.

❷ 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) to queue pending requests (PR), denoted as PR LUT in Figure 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. This satisfies Invariant 2.

❸ Each core buffers the pending write back responses in a *FIFO* queue, which Figure 3 denotes as the pending write back (PWB) FIFO. This modification cooperates with the proposed transient states to satisfy Invariant 3.

❹ Each core deploys a work-conserving TDM arbitration between the PR and PWB FIFOs. 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 write hit to a non-modified line A, it has to initiate an Upg() coherence message on the bus. With change ❶, the arbiter does not allow this Upg() message on the bus unless it is the TDM slot of the initiating core. In consequence, the write hit to A is postponed to c_i's next slot, which implements Invariant 4. Assume that during c_i's next slot, there were one or more pending requests to A from other cores that arrived before c_i's request. According to Invariant 5, c_i's write hit to A has to wait until these pending requests are serviced. Recall that PR LUT ❷ queues pending requests. If the write hit is to one of these lines, the arbiter does not elect the write hit to execute during this slot. Accordingly, Invariant 5 is fulfilled.

B. Coherence Protocol Modifications

Table I shows all possible coherence states for a cache line and the transitions between these states for PMSI. We do not make changes to the coherence states for the shared memory, and hence it is not shown. Shaded cells represent the situations where no transition occurs, while cells marked with "X" denote impossible cases under correct operation. Take for instance the case where c_i has a cache line A in state I. If c_i has a read operation to A, it issues an OwnGetS() message for A, and moves to state IS^d. On the other hand, if c_i observes an OtherGetM() message on the bus for cache line A that it has in state I, it does not make any change to A's state. Alternatively, c_i cannot have a replacement request for A, since A is originally invalid in its private cache. The

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

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

three stable states I, S, and M have the same semantics as the conventional MSI protocol as explained in Section III.

1) Removed transient states: Recall in Section III, we categorized transient states into: 1) states that indicate waiting for coherence messages to appear on the bus, and 2) states that indicate waiting for data responses. For a real-time system, the first category is not needed. On deploying a predictable bus arbitration, once a core is granted access to the bus, no other core can issue a coherence message during that slot. This is assured by Invariant 1. Accordingly, during a core slot, its coherence messages are not disrupted by messages from other cores. For example, assume that c_i has a read request to a line A that is invalid in its private cache. During c_i's slot, it issues its OwnGetS() to the bus. Since c_i is the only core issuing coherence messages to the bus, it cannot receive its data before observing its OwnGetS() on the bus. Therefore, c_i changes A's state from I to IS^d without the need to move to IS^a. By removing these transient states, PMSI has fewer states and transitions compared to the conventional MSI protocol [25]. Thus, PMSI encounters no overhead in state encoding.

2) Unmodified transient states: In contrast, the second category that denotes the waiting for data response is required in a real-time system that deploys a predictable bus arbitration and does not allow for cache-to-cache transfers. This is because if c_i issues a request to a cache line that is modified by another core c_j, c_i must wait until c_j writes back that cache line to the shared memory. If cache-to-cache transfer is not allowed, this operation consumes multiple schedule slots. Accordingly, c_i has to move to a transient state indicating that it is waiting for a data response from the memory. In Table I, these states include IS^d for a read request and IM^d for a write request. In addition, there are three other unmodified transient states in Table I, IS^{d}I, IM^{d}I, and IM^{d}S. These states indicate that the core has to take an action after receiving the data and perform the operation. Figure 4 explains the necessity of these transient states with an illustrative scenario. In Figure 4, c_1 issues a read request to A ❸, which c_0 has modified ❷. c_1 changes A's state to IS^d waiting for c_0 to write back A to shared memory. Before c_1 receives the data, c_2 requests A to modify ❹. According to Invariant 2, the memory services c_1's request to A before c_2's request. However, c_1 has to store the information that there is a pending coherence message to A that it has to respond to once

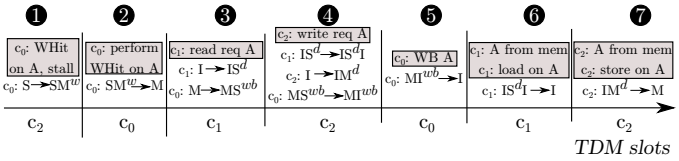


Fig. 4: Transient states example; grey boxes are events, and arrows are state transitions. Initially, c_0 has A in S.

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

TABLE II: Semantics of the proposed transient states to achieve a predictable behavior.

it completes its operation to A. This information is preserved by the transient states. For instance, during c_2's slot ❹, c_1 observes OtherGetM(A); thus, it has to move to IS^{d}I state to indicate that upon receiving the data and conducting the read operation, it has to invalidate A as c_2 will modify it ❻.

3) Proposed new states: We propose three additional transient states that are necessary to guarantee that invariants are upheld. Table II tabulates the proposed states along with their semantics. States MI^{wb} and MS^{wb} manage the write back operation. This is crucial for achieving predictability for any request to a modified cache line. For instance, in Figure 4, during c_1's slot, c_0 observes c_2's read request to A, which c_0 has modified ❹. Therefore, it marks A to be written back in the next slot by moving it to the MS^{wb} state. This indicates that c_0, in the next designated slot, will write back A to the memory and change its local copy of A to S state. Before c_0 writes back A to shared memory, it observes c_2's modify request to A ❹. As a consequence, it updates its A's state to MI^{wb}, which indicates that c_0 has to invalidate A once it performs the write back operation ❺. State SM^w is necessary to handle write hits to non-modified lines predictably. For example, in Figure 4, during c_2's slot ❶, c_0 has a write hit to A, which it has in S state. To impose Invariant 4, c_0 has to postpone this operation

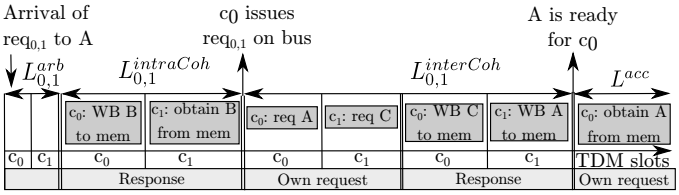


Fig. 6: Different latency components. Initially, c_0 modified B and c_1 modified A.

to its next slot. Towards doing so, it updates its A's state to SM^w to preserve the information of the upgrade request to A. In its next slot, if no other core is pending on A (Invariant 5), c_0 issues its OwnUpg(A) on the bus, performs the write to A, and moves its A to the stable state M ❷.

VII. Latency Analysis

We derive the upper bound per-request latency that a core suffers when it attempts to access the shared memory. The considered system deploys the predictable MSI protocol proposed in Section VI and a TDM bus arbitration amongst cores. We partition this latency into four components and compute the WC value of each of them. Definitions 1–5 formally define these latency components.

Definition 1: **Arbitration latency**, $L^{arb}_{i,r}$, of a request number r generated by c_i, $req_{i,r}$, is measured from the time stamp of its issuance until it is granted access to the bus. $L^{arb}_{i,r}$ is due to requests from other cores scheduled before c_i.

Definition 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. Determining the value of L^{acc} is outside the scope of this paper and existing work can be used to determine it both for LLCs [16] and DRAMs [21].

Definition 3: **Coherence latency**, $L^{coh}_{i,r}$, of a request $req_{i,r}$ generated by c_i is measured from the time stamp when c_i is granted access to the bus until it starts its data transfer. $L^{coh}_{i,r}$ occurs due to the rules enforced by the deployed coherence protocol to ensure data correctness.

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

Definition 4: **Inter-core coherence latency**, $L^{interCoh}_{i,r}$, of a request $req_{i,r}$ generated by c_i is measured from the time stamp when $req_{i,r}$ is granted access to the bus until the data is ready by the shared memory for c_i to receive in c_i's slot. $req_{i,r}$ to a line A suffers inter-core coherence latency if another core has modified or requested A to modify before c_i issued its request.

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

Figure 6 depicts the different latency components for a dual-core system with a TDM bus arbiter. c_0 issues $req_{0,1}$ to A one cycle after its slot has started; thus, it must wait for one TDM period before it accesses the bus in its next slot; hence, $L^{arb}_{0,1}$= 2 TDM slots. However, c_0's next slot is dedicated to write back responses. c_0 writes back B to shared memory since c_1 is pending for it. Consequently, $req_{0,1}$ must wait for another period before it gets an access to the bus. $L^{intraCoh}_{0,1}$ accounts for this delay and equals one TDM period. When c_0 issues $req_{0,1}$ to the bus, it turns out that c_1 has modified it. Therefore, c_0 has to wait for c_1 to update the shared memory with the new value of A. This waiting latency is the $L^{interCoh}_{0,1}$ and equals 2 TDM periods. Finally, the memory sends the data to c_0 and the transfer consumes L^{acc}, which is a single TDM slot.

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

$$WCL^{arb}_i = N \cdot S \tag{1}$$

Proof: Recall that the deployed TDM arbiter grants one slot to each core per period. Thus, the period equals to $N \cdot S$ cycles. 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 $N \cdot S$. ■

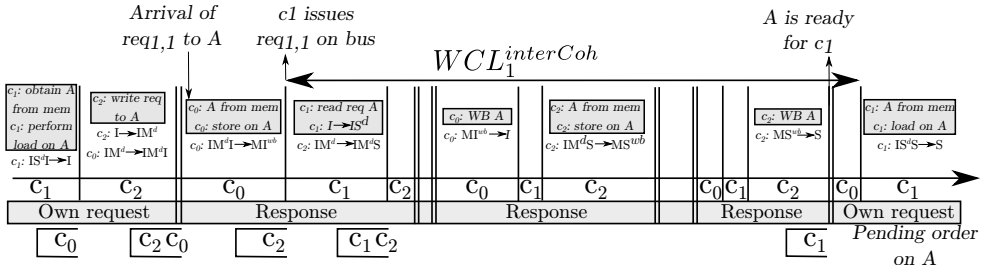
Lemma 2: The WC inter-core coherence latency, $WCL^{interCoh}_i$, occurs when a core requests a line that has been previously modified or requested to modify by all other cores.

Proof: As per Definition 4, $req_{i,r}$ to a line A suffers inter-core coherence latency if another core has modified or requested A to modify before c_i issued $req_{i,r}$. Thus, c_i has to wait until previously pending requests to A complete and the shared memory has the updated value of A before it gains an access to it. As a result, c_i suffers $WCL^{interCoh}_i$ when all other $N-1$ cores in the system requested to modify A before c_i issued its request. We prove this by contradiction. Assume that each core consumes T periods to obtain A, write to it, and update the shared memory with the new value. As a result, c_i must wait for $L_1 = (N-1) \cdot T$ periods before it accesses A. Now, assume that c_i suffers $L_2 = WCL^{interCoh}_i$ when N' cores requested to modify A before c_i issued its request, where $N' < N-1$. In this case, c_i must wait for $L_2 = N' \cdot T$ periods before it accesses A. Since $N' < N-1$, then $L_2 < L_1$. However, this contradicts the hypothesis that $L_2 = WCL^{interCoh}_i$. ■

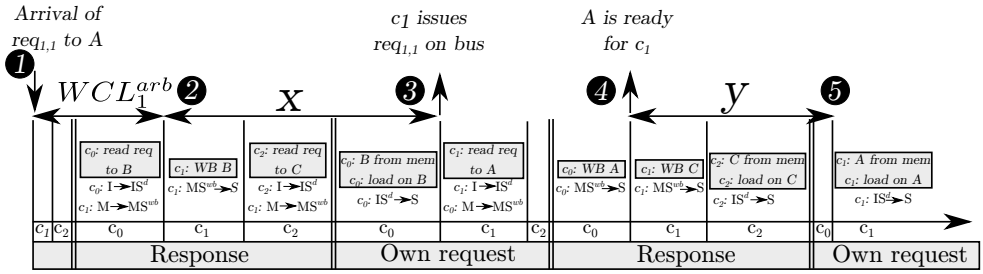
Lemma 3: $WCL^{interCoh}_i$ is calculated by Equation 2.

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

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. Figure 5a shows the WC inter-core coherence latency for c_1 in a three-core system, where c_1 waits for 4 periods from the stamp of issuing the request to the bus until its data is ready to be sent by the memory. Moreover, if $N > 2$, when the shared memory has the updated version that is ready to send to c_i,



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



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

Fig. 5: The latency bound on each interference component. Empty slots/periods do not have events that are related to c_1's latency.

c_i might have missed its slot in the current period. Therefore, it has to wait for an additional period to be able to receive A from the shared memory. In Figure 5a, the WC inter-core coherence latency of c_1 is 5 TDM periods, i.e., 15 slots. On the other hand, if $N \leq 2$, the core is guaranteed to have a slot in the same period as the data is ready at the memory. This is illustrated by Figure 6. This accounts for the second component in Equation 2. Recall that each period is $N \cdot S$ cycles. $WCL_i^{interCoh}$ is as calculated by Equation 2.

■

Lemma 4: The WC intra-core coherence latency is calculated by Equation 3.

$$WCL_i^{intraCoh} = \begin{cases} 2 \cdot N \cdot S & N > 2 \\ N \cdot S & N \leq 2 \end{cases} \quad (3)$$

Proof: There exist two cases:

1) **Case of** $N > 2$**.** A request from c_i implies two actions from c_i. First, issuing the request to the bus. Second, receiving the data from the shared memory. As a result, the worst-case intra-coherence latency occurs when each of these actions is delayed by write back responses that c_i has to conduct. Since the system deploys a work-conserving TDM between responses and own requests. Each action can encounter a maximum delay of one TDM period. Accordingly, the WC intra-coherence latency is two TDM periods or $2 \cdot N \cdot S$. Figure 5b delineates this situation. For the first action, although c_1 is granted a slot at time stamp ❷, it is a designated write back slot. Thus, c_1 does not issue its request until ❸, which is one TDM period later. For the second action, although the memory has the data ready for c_1 in its slot ❹, it is again a designated write back slot. Therefore, c_1 does not receive the data until ❺, which is one TDM period later.
2) **Case of** $N \leq 2$**.** Recall that cores are in-order such that each core can have at maximum one pending request at any instance. Hence, c_i cannot have two pending write back requests from the only other core in the system, c_j. In worst-case, c_i requests a line that is modified by c_j. Thus, it has to wait for two TDM periods because of inter-core coherence interference as per Lemma 3. 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. Figure 6 illustrates this situation.

■

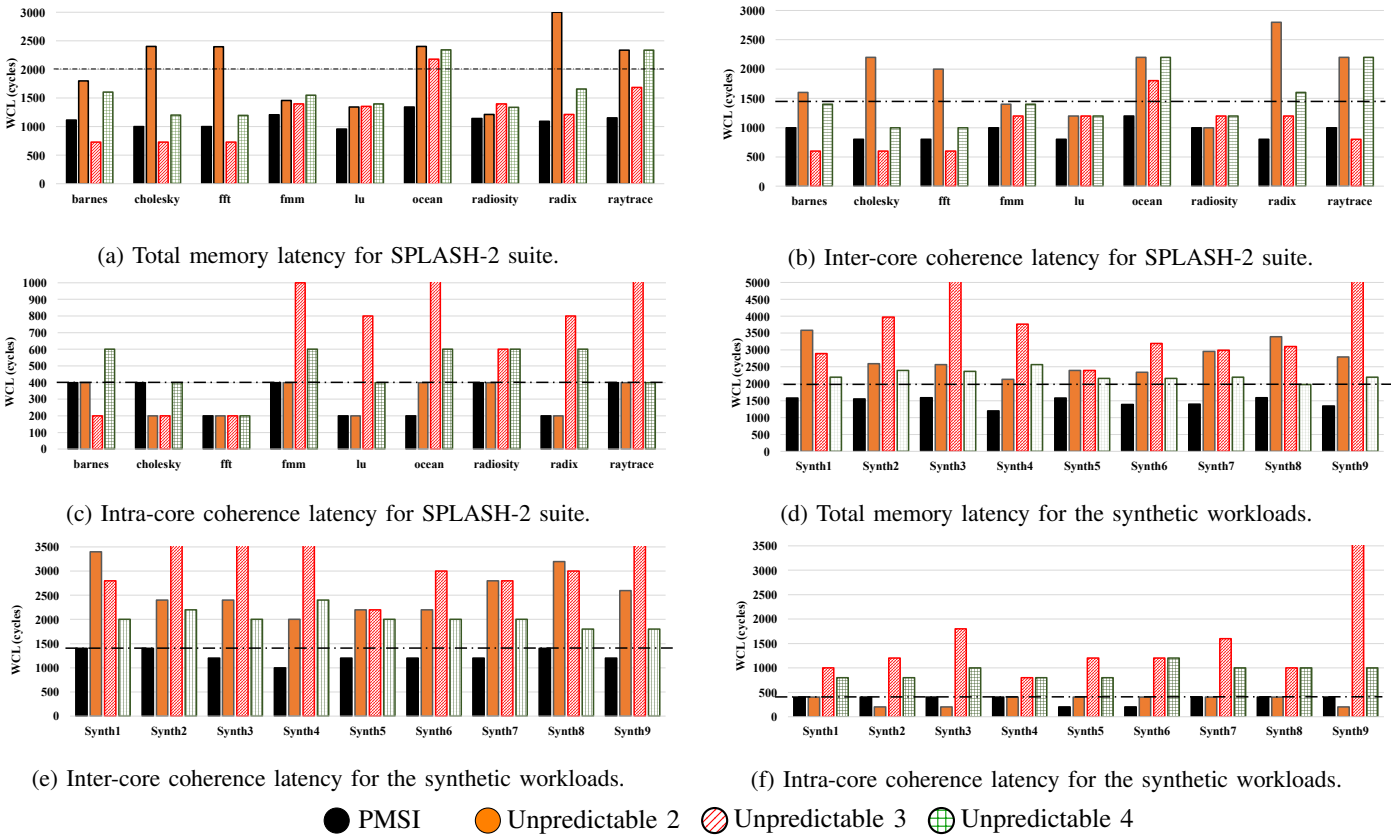
Theorem 1: The total WCL suffered by a core c_i issuing a request to a shared line A is calculated as:

$$WCL_i^{tot} = (2 \cdot N^2 + 1) \cdot S + \begin{cases} 2 \cdot N \cdot S & N > 2 \\ 0 & N \leq 2 \end{cases} \quad (4)$$

Proof: Recall that L^{acc} is fixed and equals to the slot width, S. From Lemmas 1, 3, and 4 and since WCL_i^{tot}=WCL_i^{arb}+$WCL_i^{interCoh}$+$WCL_i^{intraCoh}$+L^{acc}, WCL_i^{tot} can be calculated by Equation 4. ■

VIII. Evaluation

We integrate PMSI into the gem5 simulator [14]. We use the Ruby memory model in gem5, which is a cycle-accurate model with a detailed implementation of cache coherence events. We use a multi-core architecture that consists of x86 cores running at 2GHz. The cores implement in-order pipelines, which we find are representative of cores used in the real-time domain. Each core has a private 16KB direct-mapped L1 cache, with its access latency as 3 cycles. All cores share an 8-way set-associative 1MB LLC cache. Since the focus of this 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 [20], [21], 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



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

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

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

(d) Total memory latency for the synthetic workloads.

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

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

Fig. 7: WC latencies and the effect of unpredictability sources on them. Unpredictable i corresponds to source i in Section V. Horizontal dotted line represents the analytical bound.

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 using the *SPLASH-2* [29] benchmark suite. In addition, we use synthetic workloads to stress the WC behavior.

A. Verification

We verified the correctness of PMSI using various methods. 1) We used the Ruby Random Tester with gem5 [14] specifically to verify coherence protocols. We stressed PMSI with 10 million random requests. 2) We used carefully-crafted synthetic micro-benchmarks to cover all possible transitions and states in PMSI. This also ensures the exhaustiveness of the identified unpredictability sources and corresponding invariants. Recall that PMSI only addresses the aforementioned sources. All observed latencies conform to the bounds. If there was an unpredictability source that is not included in Section V, it should lead to unpredictable behavior (i.e., observed latencies would exceed the bound) at one or more of the transitions, which we did not observe. 3) We executed the applications in the SPLASH-2 suite on gem5 using PMSI and they run to completion. Furthermore, we check data correctness by checking the output of each application.

B. Exp.1: Bounding the Memory Latency

We study the effectiveness of PMSI to bound the delays resulting from coherence interference. We also study the effects of violating each one of the invariants on the memory latency. We use a 4-core system for our experiments. For SPLASH-2, we launch each SPLASH-2 application as four threads using four single-threaded cores, where only one application is used per experiment. Figure 7 depicts our findings. It shows the observed WC latencies for a) the total memory latency, b) the inter-coherence latency, and c) the intra-coherence latency. Since SPLASH-2 applications are optimized to minimize data sharing, they do not stress the coherence protocol. Therefore, to further stress the coherence protocol, we execute synthetic experiments using 9 synthetically-generated workloads: Synth1 to Synth9 in Figure 7. In each synthetic experiment, we simultaneously run four identical instances of one workload by assigning one instance on each core. These experiments represent the maximum possible sharing of data since each core generates the same sequence of memory requests. Figure 7 also illustrates the experimental WC latencies for these experiment. The WC arbitration latency for benchmarks in all experiments is $N \cdot S = 200$ cycles for $N = 4$ cores and slot $S = L^{acc} = 50$ cycles; hence, not shown.

Observations. 1) Figure 7 shows that for PMSI all the WC latencies are within their analytical bounds. 2) On the other hand, violating any of the invariants introduces a source of unpredictability, which results in exceeding those bounds. Moreover, for source 1, one of the cores is not able to obtain an access to a block that it requests and the program never terminates. This is the reason that Figure 7 does not show Unpredictable 1. This shows that augmenting a conventional coherence protocol with a predictable arbiter does not guarantee predictability. 3)

For a quad-core system, the latency suffered by a core due to coherence interference is $9\times$ more than the latency due to bus arbitration. The inter-core coherence interference solely contributes a latency up to $7\times$ of the arbitration latency, while the latency resulting from the intra-core coherence interference is double the arbitration latency. This provides evidence of the importance of considering the coherence latency when sharing data across multiple cores for real-time applications.

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

We compare the overhead caused by four approaches to handle data sharing in multi-core real-time systems: 1) not using private caches (*uncache-all*), 2) not caching the shared data (*uncache-shared*), 3) the proposed PMSI, 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], [8], 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 [9]. The overhead is calculated as the slowdown compared to the conventional MESI protocol. Figure 8 depicts our findings, where MSI and MESI are the conventional (unpredictable) protocols implemented as in [25].
Observations. 1) The *uncache-all* is useful when there is minimal amount of shared data being repeatedly accessed. From our experiments, we notice that data reuse is common in applications. This is the reason that *uncache-all* has the worst execution time for most applications with a geometric mean slowdown of $32.66\times$ compared to MESI. 2) Since private data does not cause any coherence interference, *uncache-shared* allows caching of only private data, while uncaching of all shared data. In Figure 8, *uncache-shared* has better performance than *uncache-all* for all applications with a geometric mean slowdown of $2.11\times$ Nonetheless, *uncache-shared* requires additional hardware and software modifications to distinguish and track cache lines with shared data, which are the same modifications required by [10]. 3) Mapping applications with shared data to the same core avoids data incoherence since these tasks share the same private cache. However, it prohibits parallel execution of these application. In consequence, for some applications (*fft*, *radix*, and *raytrace*), *single-core* achieves better performance compared to *uncache-shared*, while for other applications, it exhibits lower performance. This is dependent on several factors such as the memory-intensity of the application and the ratio of shared to non-shared data. Overall, *single-core* achieves a geometric slowdown of $2.67\times$. 4) Finally, PMSI achieves better performance compared to all other predictable approaches for all benchmarks, except for *radiosity* and *raytrace*. PMSI achieves improved performance of up to $4\times$ the best competitive approach, *uncache-shared*, with a geometric mean slowdown of $1.46\times$ in performance compared to MESI.

Upon analyzing *radiosity* and *raytrace*, we found that both do not show considerable reuse of the shared data. Shared lines in a core's private cache are often invalidated because of other cores before they are accessed again. Therefore, uncaching these cache lines achieves better performance. since the shared memory has the most updated value at all time instances; thus, cores do not suffer coherence interference.

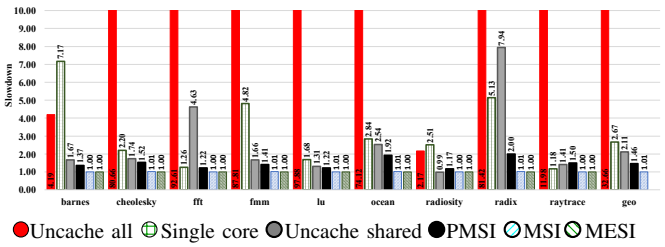


Fig. 8: Execution time slowdown compared to MESI protocol.

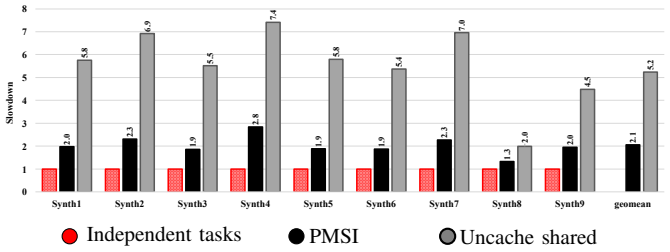


Fig. 9: Slowdown in the execution time of different approaches compared to ideal scenario.

D. Exp.3: Comparing to the Ideal Scenario

It is desirable to minimize the coherence interference, while allowing tasks to simultaneously access shared data. Optimally, the coherence interference equals zero. Although this is attainable only if all running tasks are independent so as they do not share data, it can be used as the ideal metric to compare different approaches to it.
Methodology. In this experiment, we study the slowdowns resulting from the proposed PMSI and the *uncache-shared* approach compared to the ideal case, *independent-tasks*. We stress both PMSI and *uncache-shared* by using our synthetic workloads similar to Exp.1. In *independent-tasks*, for each application, we simultaneously run three other applications in the three other cores, which do not share data with the current application. In *uncache-shared*, we simultaneously run four identical instances of each application, one instance on each core. Since these applications share all their memory data, *uncache-shared* and *uncache-all* are equivalent approaches; thus, we do not consider the *uncache-all* case. We delineate the results of these experiments in Figure 9.
Observations. 1) Since the data sharing is maximum for *uncache-shared*, all memory requests have to access the shared memory suffering from L^{acc} latency. Accordingly, *uncache-shared* suffers from severe slowdowns compared to the ideal case, *independent-tasks*. In Figure 9, its execution time ranges from $2\times$ to $7.4\times$ compared to *independent-tasks*, with a geometric mean of $5.2\times$. 2) On the other hand, PMSI allows cores to cache the data (both private and shared) to their private caches, which improves the overall performance. PMSI's execution time ranges from $1.3\times$ to $2.8\times$ compared to *independent-tasks*, with a geometric mean of $2.1\times$. This illustrates the importance of deploying a coherence protocol to manage data sharing, while decreasing performance overheads.

E. Exp.4: Scalability

Methodology. We study the impact of each latency type upon increasing the number of cores. We run one instance of

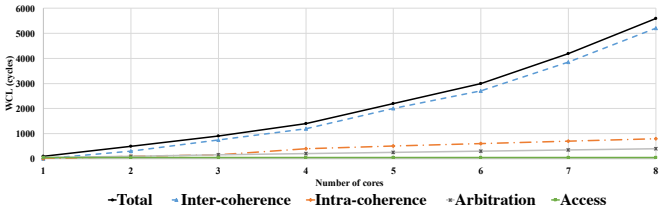


Fig. 10: Latencies with different number of cores.

the Synth1 workload on the core under consideration, c_0, and sweep the number of co-running cores from 0 to 7. Each co-running core also executes the Synth1 workload. Figure 10 shows the experimental WC latencies of c_0.

Observations. Clearly, increasing the number of cores, the increasing rate in the inter-coherence latency is much larger than that of the arbitration and intra-coherence latencies. This aligns with the analysis in Section VII. The inter-coherence latency is a quadratic function, while both the arbitration and intra-coherence latencies are linear functions in the number of cores. Moreover, the increasing rate of the intra-coherence latency is double that of the arbitration-latency. Figure 10 shows that increasing the number of cores, the coherence-latency dominates the total memory latency. This emphasizes the importance of carefully considering the coherence latency impact on multi-core real-time systems upon allowing for simultaneous accesses to shared data.

IX. 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 protocol as an example. Towards this target, we propose a set of novel transient states as well as minimal architecture requirements. We experiment using the SPLASH-2 benchmark suite and worst-case oriented synthetic workloads.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd for their valuable feedback and suggestions.

References

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

[2] M. Paolieri *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, pp. 57–68.

[3] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *Ninth European Dependable Computing Conference*, 2012, pp. 132–143.

[4] S. Schliecker *et al.*, "System level performance analysis for real-time automotive multicore and network architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 979–992, 2009.

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

[6] G. Gracioli *et al.*, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, pp. 32:1–32:36, 2015.

[7] D. Hardy *et al.*, "Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches," in *30th IEEE Real-Time Systems Symposium (RTSS)*, 2009, pp. 68–77.

[8] B. Lesage *et al.*, "Shared data caches conflicts reduction for WCET computation in multi-core architectures." in *18th International Conference on Real-Time and Network Systems (RTNS)*, 2010, p. 2283.

[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, pp. 194–204.

[10] A. Pyka *et al.*, "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, pp. 107–114.

[11] "Predictable cache coherence for multi-core real-time systems." [Online]. Available: https://git.uwaterloo.ca/caesr-pub/pmsi

[12] D. Hackenberg *et al.*, "Comparing cache architectures and coherency protocols on x86-64 multicore smp systems," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 413–422.

[13] M. E. Thomadakis, "The architecture of the Nehalem processor and Nehalem-EP SMP platforms," *Resource*, vol. 3, p. 2, 2011.

[14] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, pp. 1–7, 2011.

[15] M. Hassan and H. Patel, "Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–11.

[16] B. C. Ward *et al.*, "Making shared caches more predictable on multi-core platforms," in *25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013, pp. 157–167.

[17] V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *Proceedings of the 45th Annual Design Automation Conference (DAC)*, 2008, pp. 300–303.

[18] M. Schoeberl *et al.*, "Towards time-predictable data caches for chip-multiprocessors," in *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 2009, pp. 180–191.

[19] J. Reineke *et al.*, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011, pp. 99–108.

[20] Z. P. Wu *et al.*, "Worst case analysis of dram latency in multi-requestor systems," in *IEEE 34th Real-Time Systems Symposium (RTSS)*, 2013, pp. 372–383.

[21] M. Hassan *et al.*, "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, pp. 307–316.

[22] D. B. Kirk and J. K. Strosnider, "SMART (strategic memory allocation for real-time) cache design using the MIPS R3000," in *Proceedings 11th Real-Time Systems Symposium (RTSS)*, 1990, pp. 322–330.

[23] B. Lesage *et al.*, "PRETI: Partitioned real-time shared cache for mixed-criticality real-time systems," in *Proceedings of the 20th International Conference on Real-Time and Network Systems (RTNS)*, 2012, pp. 171–180.

[24] J. Bin *et al.*, "Studying co-running avionic real-time applications on multi-core COTS architectures," in *Embedded Real Time Software and Systems conference*, vol. 15, 2014.

[25] D. J. Sorin *et al.*, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[26] A. Cortex, "Cortex-A9 MPCore," *Technical Reference Manual*, 2009.

[27] N. Kurd *et al.*, "Next generation Intel® core micro-architecture (nehalem) clocking," *IEEE Journal of Solid-State Circuits*, pp. 1121–1129, 2009.

[28] H. Yun *et al.*, "Parallelism-aware memory interference delay analysis for COTS multicore systems," in *27th Euromicro Conference on Real-Time Systems (ECRTS)*, 2015, pp. 184–195.

[29] S. C. Woo *et al.*, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings 22nd Annual International Symposium on Computer Architecture (ISCA)*, 1995, pp. 24–36.