2019 IEEE Real-Time Systems Symposium (RTSS)

Enabling Predictable, Simultaneous and Coherent Data Sharing in Mixed Criticality Systems

Nivedita Sritharan *, Anirudh Mohan Kaushik *, Mohamed Hassan †, and Hiren Patel *
* Department of Electrical and Computer Engineering, University of Waterloo, Ontario, Canada, {first.last}@uwaterloo.ca
† Department of Electrical and Computer Engineering, McMaster University, Ontario, Canada, mohamed.hassan@mcmaster.ca

***Abstract*—Emerging embedded systems deployed in the automotive and avionics domains execute applications with different criticalities, comprising what is known as Mixed Criticality Systems (MCS). Applications in MCS often share data between tasks (coming from sensors for instance). Data sharing is challenging because it can lead to increased response times or even unpredictable behaviors if not carefully addressed. Therefore, several prior works in MCS either assumed that tasks do not share data or disallowed it by design. Recent solutions attempt to mitigate the effects of data sharing, albeit by introducing new restrictions on the system either by prohibiting applications from caching shared data or prohibiting the operating system from running tasks with shared data in parallel. We find these solutions also to have limited applicability as they deteriorate system schedulability and prohibit simultaneous access to shared data. In this paper, we propose PENDULUM: a time-based cache coherence protocol to enable simultaneous and predictable access to shared data in MCS. Our evaluation shows that PENDULUM, achieves flexibility and better performance compared to existing solutions, while maintaining system predictability.**

***Index Terms*—Mixed-criticality systems, multi-core platforms, cache coherence protocols**

I. Introduction

Embedded systems are ubiquitous nowadays. They enable many domains including: automotive [1], [2], avionics [3], [4], and healthcare [5], [6]. Embedded systems in these domains execute tasks of different criticalities, comprising what is known as Mixed Criticality Systems (MCS). In MCS, both critical (Cr) and non-critical (nCr) tasks execute on the same computing platform, and hence, share hardware resources. Although there has been a recent focus on the predictable management of these resources to satisfy requirements of MCS [7]–[11], prior works either assumed that tasks do not share data or disallowed it entirely by design. This is because sharing data if not carefully handled can lead to unpredictable behaviors [12], [13]. Recent works made a similar observation [12]–[14] and explored solutions to enable data sharing among tasks [12], [15]–[18]. Our study of these solutions [12], [15]–[18] show that they ease the constraint on shared data, but at the expense of introducing new constraints on the system. For instance, cache bypassing [15], [16] prohibits shared data from being cached in private caches of cores, while shared data aware scheduling [12], [17], [18] imposes restrictions on the task-to-core mapping and scheduling. These solutions prohibit simultaneous access to shared data by multiple tasks meaning multiple cores cannot cache and access shared data in their private caches on the same time. This leads to performance degradation of the system, while shifting the problem of shared data to the software level. Bypass techniques [15], [16] require modifications to legacy software and extensions to the instruction set architecture to encode bypass decisions per memory instruction. Shared data aware techniques [12], [17], [18] require the operating system scheduler to take into account shared data information upon scheduling tasks on cores. Authors in [13] enable simultaneous access to shared data in the context of real-time systems by introducing a predictable cache coherence protocol called PMSI. However, PMSI [13] is not designed for MCS as it services all cores both Cr and nCr equally regardless of their criticality. Therefore, if the coherence protocol developed in [13] is deployed on a MCS arbiter (e.g. using fixed-priority arbitration to prioritize Cr cores), it can lead to unpredictable latencies as we discuss in Section II. This work addresses the problem of data sharing in multi-core MCS by introducing PENDULUM. PENDULUM is a criticality-aware coherence protocol that enables simultaneous, coherent, and predictable access to shared data in MCS. Towards this target, we make the following contributions.

Contributions. (1) We study the limitations of the state-of-the-art techniques that allow for shared data in MCS and discuss the causes of these limitations (Section II). (2) We enable tasks in MCS to simultaneously share and access data. This is achieved by introducing PENDULUM, a criticality-aware cache coherence protocol that orchestrates access to shared data from both Cr and nCr cores. (3) With the existence of shared data, we bound the worst-case memory latency (WCL) suffered by any Cr core through timing analysis (Section VI). (4) We equip PENDULUM with configurable timers that reduce cache line invalidations and increase system performance. By configuring these timers, the system designer has the ability to address the trade-off between reducing the WCL of Cr cores and increasing average performance of nCr cores, which are usually conflicting requirements in MCS. We study the effects of those timers and provide guidelines on selecting their values for various design use-cases (Section V-A). (5) We conduct a detailed evaluation that compares PENDULUM with the state-of-the-art techniques using both SPLASH-2 benchmarks, which are representative for applications with shared data as well as synthetic benchmarks (Section VIII). (6) Finally, we release the implementation of PENDULUM as a step towards enabling the research community to ex-

2576-3172/19/$31.00 ©2019 IEEE
DOI 10.1109/RTSS46320.2019.00045

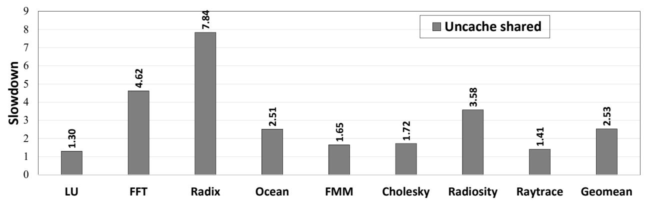


Fig. 1: The slowdown of uncaching the shared data.

plore predictable data sharing mechanisms without introducing scheduling constraints or software modifications. PENDULUM can be found at https://git.uwaterloo.ca/caesr-pub/pendulum.

II. Motivation: The State of the Art

With the rise of using multi-core platforms for real-time systems, predictably managing shared hardware resources in these platforms became a necessity. Therefore, several recent research efforts were proposed to provide predictable access to shared interconnects (e.g. [9], [19], [20]), shared caches (e.g. [21]–[23]), and shared DRAMs (e.g. [10], [11], [24]). A subset of these efforts took criticality into account targeting MCS such as [7]–[11]. These works focus on predictably resolving timing interference, i.e. bounding the latency overheads resulting from other competing cores that access these shared resources. However, another source of interference is data interference, which results from data sharing among cores. Two common approaches to avoid such interference are (1) assume tasks do not share data or (2) disable data sharing by design. However, both these approaches disable communication between multiple cores in the multi-core system. As a result, different tasks or even jobs of the same parallel task running on multiple cores cannot share data. To address this limitation, recent works have investigated solutions to enable data sharing, which we describe in the following subsections.

A. *Uncaching Shared Data*

This approach disables caching of shared data in private cache hierarchies to provide predictable access to shared data [15], [16]. This comes at the expense of both degraded average-case performance and high WCL since any access to this shared data has to go to the shared memory. To illustrate the effect of this limitation, we plot in Figure 1 the slowdown resulting from uncaching the shared data as compared to allowing to cache this data using the Modified-Shared-Invalidate (MSI) coherence protocol for the SPLASH-2 benchmarks. The slowdown reaches up to $7.84\times$ for the Radix benchmark. The detailed setup is explained in Section VIII. Additionally, this approach requires modifications to legacy software and extensions to the instruction set architecture to encode bypass decisions per memory instruction.

B. *Shared Data Aware Scheduling*

The second approach is to schedule tasks at the operating system level such that interference due to shared data is mitigated [12], [17], [18]. This approach allows different tasks to share data; nonetheless, it puts constraints on how they are scheduled by the system scheduler. For instance,the solution described in [12] maps all tasks with shared data to same core. The solution in [18] focuses on automotive domain and is limited to a specific task model, where a task can be statically composed of three distinct phases: read, execute, and write phases. Then, tasks are scheduled such that overlap between memory phases (read and write) of different tasks is strictly prohibited; thus, shared data is not simultaneously accessed.

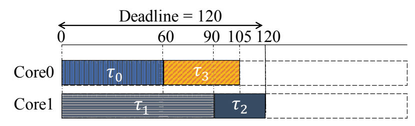
We argue that this approach deteriorates the system schedulability. To illustrate the limitations of this approach, Figure 2 delineates the scheduling of four tasks, τ_0–τ_3 with execution times of 60, 90, 45, and 30, respectively. All four tasks have the same deadline of 120 and are released at time 0. Task τ_1 shares data with both τ_0 and τ_3, but there is no shared data between τ_0 and τ_3. Task τ_2 does not share data with other tasks. The system has two cores, Core0 and Core1. In Figure 2a, the scheduler does not take into account any shared data constraints. It schedules tasks τ_0 and τ_3 in Core0 and tasks τ_1 and τ_2 in Core1. In this schedule, all tasks meet their deadlines. In Figure 2b, the scheduler applies the constraint of prohibiting tasks with shared data from running in parallel. Thus, task τ_1 cannot run in parallel with τ_0 and τ_3. Under this constraint, there is no feasible schedule where all tasks can meet their requirements. For the schedule shown in Figure 2b, task τ_1 finishes at time 150 and misses its deadline. The scheduler in Figure 2c adopts the constraint of mapping tasks with shared data to the same core. Thus, it maps tasks τ_0, τ_1, and τ_3 to Core0. Again, there no feasible schedule that can meet the requirements of all tasks. Task τ_1 in Figure 2c misses its deadline by finishing at time 195. From this example, imposing constraints on system scheduler to mitigate shared data interference deteriorates system schedulability and can deem task sets unschedulable, which otherwise can be scheduled.

Based on this discussion, we strongly believe that a new solution is needed to enable data sharing in MCS without imposing new constraints on the system. The solution we propose handles shared data, while not requiring alterations to either the application software and the system scheduler by enforcing cache coherence at the hardware level. We next briefly discuss the related work in cache coherence.

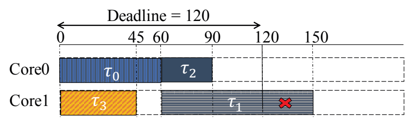
C. *Cache Coherence*

Cache coherence enables cores in multi-core architectures to have simultaneous access to shared data, while maintaining a correct and updated view of this data for all cores. A read to a shared cache line returns the result of the most recent write to this line. We cover the background of the detailed operation of cache coherence protocols in Section III, while focusing on the high-level view and the related works in this section.

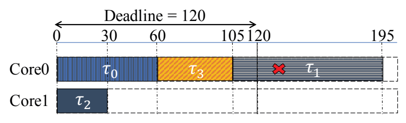
1) Cache Coherence in General-Purpose Computing: Cache coherence has been extensively studied in the context of general-purpose computing [25]. It is the mainstream solution for handling shared data in multi-core platforms [26]. There are two classes of cache coherence protocols: (1) snooping bus-based protocols and (2) directory-based protocols [27]. In



(a) No shared data constraints on scheduling. Task set is schedulable.



(b) Tasks with shared data cannot run in parallel. Task set is unschedulable.



(c) Tasks sharing data have to be on same core. Task set is unschedulable.

Fig. 2: Limitations of the shared-data aware scheduling approach: shared data induced constraints on task scheduler.

snooping bus-based protocols, cores broadcast their memory requests to all cores on a snooping bus. However, broadcasting limits the scalability of snooping bus-based protocols for systems with large number of cores. In directory-based protocols, cores send coherence messages to a directory that holds sharer information about the requested data, and the directory responds with appropriate actions to ensure a coherent view of the requested data across cores. These protocols offer a trade-off that improves scalability with increasing number of cores at the expense of larger request latencies due to additional communication between the directory and the requesting cores.

2) Time-Based Cache Coherence: With increasing core count, the number of coherence messages transmitted on the network also increases resulting in increased network traffic and increased request latencies to shared data. To mitigate this impact of increasing core count on the request latencies to shared data, time-based cache coherence protocols have been proposed for multi-core systems with large core counts (many-core systems) [28]–[33]. The key idea behind time-based cache coherence is that once a core obtains a copy of a shared cache line in its private cache, it maintains its validity for a certain period of time regardless of the actions of the other cores, and self invalidates this copy after the time period. Self invalidation removes the need to transmit certain coherence messages resulting in reduced network traffic.

For MCS, we find that the timers in time-based cache coherence protocols, which track the duration of cache lines held by cores in their private caches, provide a natural way to enable bounded access latencies to shared data and criticality awareness. Hence, in this work, we take inspiration from time-based cache coherence protocols, and design PENDULUM, a time-based snooping bus-based cache coherence protocol for MCS. Prior time-based cache coherence protocols dynamically adapted the timer configurations during application execution in order to deliver better application performance. On the other hand, PENDULUM uses deterministic timer configurations that do not change during application execution in order to meet the predictability requirements of MCS, while providing as much performance as possible for nCr requests on a best-effort basis.

3) Cache Coherence for Real-Time Systems: Recently, Hassan et al. [13] proposed the PMSI cache coherence protocol, which is a predictable cache coherence protocol for multi-core real-time systems. In this work, they showed that deploying a predictable cache coherence protocol provided significant performance advantages over the aforementioned approaches. However, the PMSI cache coherence protocol assumes that all cores are of the same criticality level (Cr). As a result, deploying the PMSI cache coherence protocol on a MCS will result in requests from Cr cores to be subjected to significant coherence interference from nCr cores.

The need for a criticality-aware coherence protocol. An important observation we make in this work is that deploying a predictable, yet non-criticality-aware coherence protocol such as PMSI in a MCS will hinder the required independence between Cr and nCr applications mandated by standards such as the IEC-61508 [34]. This is true even when Cr applications are offered a higher priority. The reason for this is that a Cr core can gain access to the arbiter and issue a memory request; however, this request can be to a cache line that is either owned or previously requested by a nCr core. In this case, the Cr core has to wait for the nCr core based on coherence protocol rules (e.g. Invariant 2 in [13] that ensures the order of access to the same cache line based on the issuance time). Worse still, the Cr request may have unbounded latency because of this dependence on the nCr behaviour. This occurs if a Cr request has to wait for a nCr request to same cache line, which in turn is not guaranteed a bounded latency to finish.

On the other hand, PENDULUM by construction considers the mixed-criticality nature as a first-class principle. Thus, it provides tight latency bounds for Cr cores, while providing as much performance as possible for the nCr ones. In addition, unlike PMSI, PENDULUM is a time-based coherence protocol, which utilizes timers to reduce invalidations and coherence misses. Further, it provides configurable timers to enable the designer to tune the system for specific performance/bounds requirements, which are dependent on the application.

III. Cache Coherence Protocols: A background

A cache coherence protocol implements a set of rules that ensure data correctness. This set identifies states that denote the read and write permissions of each cache line in the core's private cache. Transitions between these states occur due to either an activity by the core itself (for instance, a load or store instruction) or activities of other cores in the system on the same cache line (such as a load or store miss to this cache line). The MSI cache coherence protocol is a standard cache coherence protocol that several modern cache coherence protocols are based upon such as MESIF and MOESI protocols [27]. Therefore, we use MSI in this section to explain the basics of coherence protocols. MSI has three *stable states*. The semantics for each of these states are as follows: (1) Invalid (I) indicates that the cache line does not have valid data. (2)

TABLE I: MSI protocol. msg/state denotes that a core issues the message msg, and moves to coherence state state.

State	Current core			Other cores		
	Load	Store	Replacement	Load Miss (OtherGetS)	Store Miss (OtherGetM)	Other Upg
Invalid (I)	Issue GetS/S	Issue GetM/M	–	–	–	–
Shared (S)	Hit	Issue Upg/M	/I	–	/I	/I
Modified (M)	Hit	Hit	Issue PutM/I	Send Data/I	Send Data/I	–

Modified (M) represents that the core has modified the cache line data; hence, it *owns* the cache line, and has the most up-to-date data. Only one core can have a cache line in the M state. (3) Shared (S) identifies that the cache line was read, but not modified. Multiple cores may have the same cache line in the S state. This allows read hits in their respective private caches. Allowing multiple cores to have a cache line in the S state, and only one core to have a cache line in the M state is referred to as maintaining the single writer multiple reader (SWMR) invariant.

Transitions between states occur by exchanging coherence messages among cores and between cores and shared memory. Table I illustrates transitions among stable states of the MSI protocol. For instance, in Table I, when a core has a load (store) request to a cache line that is currently in I state, it issues a GetS (GetM) message on the bus, waits until it receives its data and then moves to the S (M) state. The GetS (GetM) message issued by this core on the bus is going to be observed by other cores as OtherGetS (OtherGetM) message. On the other hand, if a core wants to evict (replace) a cache line that is in the M state, it has to issue a PutM message to write back the data to the shared memory before it moves to the I state.

IV. System Model

We consider a multi-core MCS that consists of a set of both Cr and nCr tasks running on N cores. Each core can run any number of tasks; however, for sake of simplifying the discussion, we assume that all tasks mapped to a specific core are either Cr or nCr. Accordingly, cores can be classified into two sets: a set of N_{Cr} critical cores running Cr tasks and a set of N_{nCr} non-critical cores running nCr tasks, where $N = N_{Cr} + N_{nCr}$. It is worth noting that PENDULUM can be seamlessly deployed in MCS where a core runs both Cr and nCr tasks. However, in this case, the bus arbiter needs to consider tasks not cores [35], and it must track the criticality of the task currently running by each core such that criticality-dependent states of PENDULUM (Section V-D) will depend on the task identification number (ID) instead of the core ID. This is true because there is no coherence interference between tasks on same core since they share the same private caches. Other sources of interference that can arise from sharing the same core (e.g. cache thrashing) are outside the scope of this paper, and can be resolved using existing solutions that are orthogonal to PENDULUM such as cache locking, or coloring [14]. Aside from this assumption, we do not require any restrictions on how tasks are mapped to cores or scheduled. This allows PENDULUM to easily integrate with various existing task scheduling techniques targeting multi-core platforms. Each core has its own private cache(s), which is usually the L1 cache and shares a lower-level of memory with all other cores, which can be the on-chip last-level cache (LLC), the off-chip main memory, or both. Cores share a common interconnect (e.g. a bus) connecting private caches and the shared memory. Private caches can also communicate with each other, which allows for a cache-to-cache data transfer. This is the common architecture in commercial-off-the-shelf (COTS) platforms as it increases system performance. For the timing analysis, we assume that cores are in-order and allow for only one outstanding memory request.

PENDULUM does not impose restrictions on how the timing interference on the shared memory is resolved, whether it is the LLC or the main memory. For simplicity of exposition, we conduct the analysis assuming accesses to the shared memory are managed through a fixed-priority arbitration, where Cr cores have a higher priority than nCr cores. Requests from different Cr cores are arbitrated using a work-conserving time-division multiplexing (TDM) scheduler, while arbitration among requests from different nCr cores uses round robin (RR). This arbiter uses a TDM-based scheduler, where each Cr core is granted one slot each TDM period. nCr cores are granted access to the shared bus only during TDM slots where none of the Cr cores have a pending request. We call these slots *slack slots*. Selecting which nCr core is granted a slack slot is determined by RR arbitration. Assuming a specific arbitration scheme helps in example illustrations and in conducting timing analysis to derive bounds for a request to the shared memory in the presence of data sharing. Nonetheless, existing work on arbitrating access to shared memories either for caches or for DRAM is orthogonal to this work and can be integrated with PENDULUM to manage shared data. Moreover, prioritizing Cr over nCr cores is a common approach in MCS since Cr cores are time-sensitive and require tight timing bounds, while nCr cores are generally performance-oriented. Cores can share data among each other to allow for communication across threads of the same task or across tasks. PENDULUM is a hardware cache coherence protocol that enables data sharing without requiring modifications to software. Accordingly, it works for MCS with no shared data, MCS with shared data among the same criticality level only (between Cr tasks for instance), and MCS with shared data among different criticality levels.

V. Proposed Solution: PENDULUM

We introduce the details of the PENDULUM coherence protocol, and explain how it enables coherent and simultaneous sharing of data while taking into account unique design principles of MCS. Due to space constraints, we explain in detail only the coherence states that are either introduced or

TABLE II: The four timer values of PENDULUM. Each core based on its criticality has two timer fields per cache line.

Timer	Owner	Requesting core	Effect on Cr cores		Effect on nCr cores	
			Average performance	WCL	Average performance	WCL
Timer(Cr,Cr)	Cr	Cr	Dependent on application	Increases	No effect	No effect
Timer(Cr,nCr)	Cr	nCr	Increases	No effect	Decreases	Increase
Timer(nCr,Cr)	nCr	Cr	Decreases	Increases	Increases	No effect
Timer(nCr,nCr)	nCr	nCr	No effect	No effect	Dependent on application	Increases

modified by PENDULUM, while we briefly explain the states that PENDULUM directly inherits from MSI-based protocols without modification. For a detailed explanation of all the transient states of MSI-based protocols, we refer the reader to a technical report [36]. The main motivation in modifying the coherence protocol is to achieve two design goals: (1) time-based coherence, and (2) criticality-aware coherence. We next detail the realization of these two goals.

A. Timers

We maintain two timer fields per cache line for each core. Values of these fields are selected from four possible timer values based on the criticality of the owner and requestor cores; thus, timer values are expressed as Timer(Owner,Requestor). The basic idea of these timers is that the Owner holds a valid copy of the cache line for a period of time equal to the timer value before it invalidates itself to provide access to the Requestor. If there is no Requestor to that cache line when the timer expires, the Owner does not invalidate its data copy, and the timer value is replenished to allow the Owner to retain the cache line for another timer period. As Table II illustrates, a cache line in a Cr core has timers Timer(Cr,Cr) and Timer(Cr,nCr), while a cache line in a nCr core has timers Timer(nCr,Cr) and Timer(nCr,nCr). The timer counters are configurable, and timer values are use-case specific and depend on the characteristics of the running tasks. Next, we discuss the impact of each timer to help the system designer in determining the suitable timer values. We discuss this impact on both average performance as well as per-request WCL of Cr and nCr cores. The timing analysis to compute the WCL is discussed in Section VI. The discussion of the average performance uses the cache average access time as defined by Equation 1, where $\%hitRate$ is the application's L1 cache hit rate, t_{hit} is the L1 cache hit access time, and t_{miss} is the miss penalty of the L1 cache.

$$t_{avg} = \%hitRate \cdot t_{hit} + (1 - \%hitRate) \cdot t_{miss} \quad (1)$$

1) Timer(Cr,Cr): With increasing Timer(Cr,Cr), the Cr requestor has to wait longer to receive data for a requested cache line that is owned by another Cr core. As a result, t_{miss} increases, which in turn increases both the WCL and t_{avg} of Cr cores. However, increasing Timer(Cr,Cr) allows a Cr owner core to have multiple hit accesses to the owned cache line during this period. This would increase $\%hitRate$, which in turn decreases t_{avg}. Therefore, the overall effect of increasing Timer(Cr,Cr) on average performance of Cr cores is application dependent as it depends on how many requests are issued to a certain cache line within a specific time window. Timer(Cr,Cr) has no effect at all on nCr cores.

Guideline. If minimizing Cr's WCL is the most important target, Timer(Cr,Cr) should be set to its minimum value.

2) Timer(nCr,nCr): Similar to Timer(Cr,Cr), increasing Timer(nCr,nCr) increases the WCL of a memory request from a nCr core, while its effect on the average-case performance of nCr cores is application dependent. Timer(nCr,nCr) has no effect at all on Cr cores.

Guideline. Tasks running on nCr cores do not require strict timing guarantees, and benefit from improved average-case performance. Hence, a task running on an nCr core can benefit from a high Timer(nCr,nCr) value if it exhibits temporal locality in its memory access pattern.

3) Timer(Cr,nCr): Since the requestor in this case is nCr, Timer(Cr,nCr) has no effect on the WCL and on the t_{miss} of the Cr cores. Increasing Timer(Cr,nCr) increases the average-case performance of Cr cores, since it increases the $\%hitRate$. On the other hand, it increases the WCL of nCr cores and deteriorates their average performance. This is because it increases t_{miss} of nCr cores.

Guideline. Configuring Timer(Cr,nCr)'s value enables the designer to decide the trade-off between average-case performance demands from nCr cores from one side and the Cr cores from the other side. For instance, if Cr applications are latency-sensitive with low average-case demands, while nCr ones require high average performance, Timer(Cr,nCr) can be set to its minimum value.

4) Timer(nCr,Cr): Timer(nCr,Cr) is the counterpart of Timer(Cr,nCr). Accordingly and following a similar rationale, increasing Timer(nCr,Cr) increases the average-case performance of nCr at the expense of increasing the WCL and deteriorating the average performance of Cr cores.

Guideline. Setting Timer(Cr,nCr)'s value enables the designer to address the trade-off between average performance demands from nCr cores from one side and the WCL demands from Cr cores from the other side.

After detailing the operation of PENDULUM's configurable timers, we now discuss the coherence states of PENDULUM. Table III shows all the transient states and transitions of PENDULUM. In addition to the three MSI coherence messages discussed in Section III, PENDULUM uses three other messages: (1) SelfInv, which is issued by a core that wants to downgrade a cache line from shared to invalid state (e.g. because of replacement). (2) AllInv, which is a message broadcasted by the shared memory when all sharers have self-invalidated. This coherence message is necessary to inform a pending store to a shared data that it is safe to proceed with

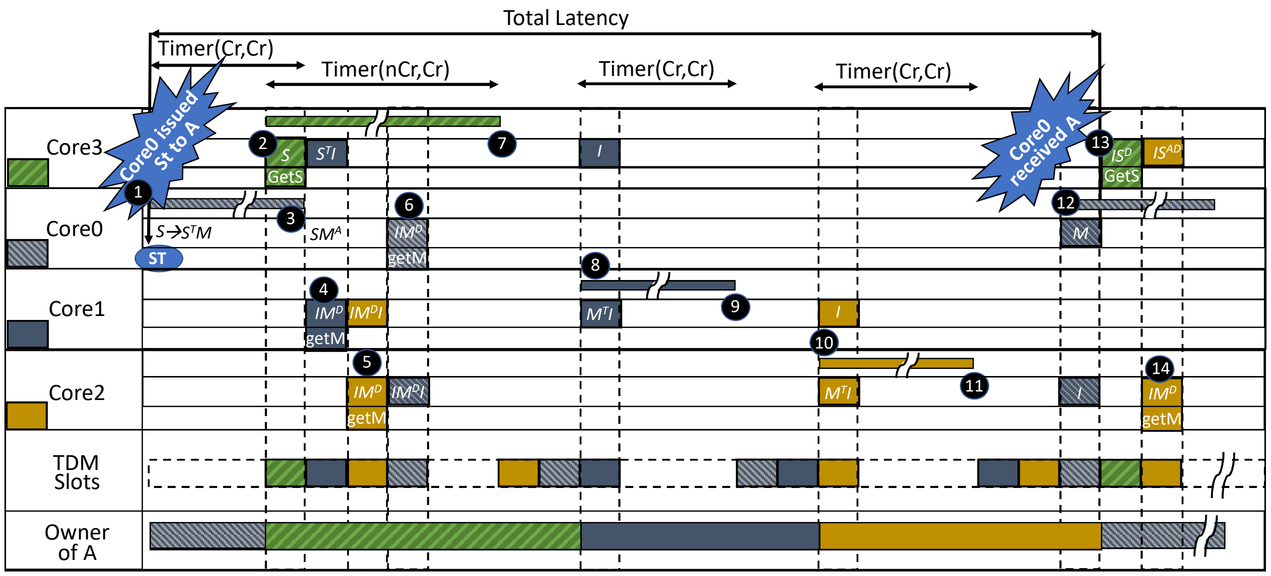


Fig. 3: An example of PENDULUM's operation. Cores0–2 are Cr, while Core3 is nCr. All requests are to same cache line A.

the store request. (3) SendData, which is sent by a core that is ready to send a cache line to a requestor. We divide the coherence states in Table III into three categories: (1) states that are inherited from the MSI-based protocols (Section V-B), (2) timer-related states (Section V-C), and (3) states that enable criticality-awareness (Section V-D).

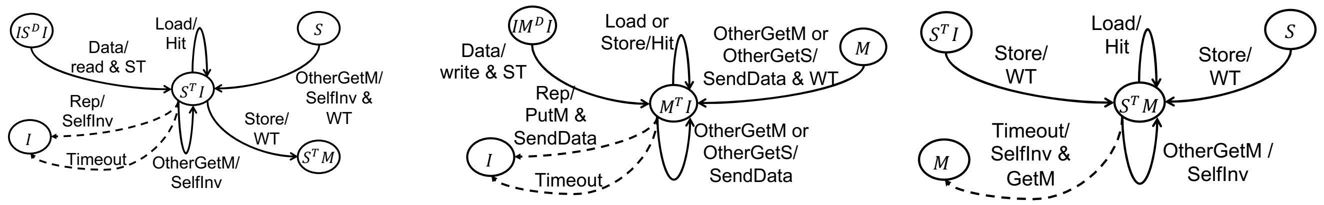
B. MSI-based Coherence States

We briefly explain the MSI-based transient states and for a detailed background on these states, we refer the reader to the technical report in [36]. MSI-based states in Table III can be classified into two types: (1) transient states to indicate the waiting for coherence messages (IS^{AD}, IM^{AD}, SI^{A}, SM^{A}, and MI^{A}), and (2) transient states to indicate the waiting for data messages (IS^{D}, IM^{D}, IS^{D}I, and IM^{D}I). The first type indicates that the core issued a coherence message, but did not observe it yet on the bus. For instance, IS^{AD} (IM^{AD}) means that a load (store) request to a cache line that was in I state is issued, but the OwnGetS (OwnGetM) is not observed yet. The second type indicates that the core observed its coherence message on the bus, but is waiting to receive the requested data. For instance, IS^{D} (IM^{D}) indicates that a load (store) request was issued to a cache line that was in I state, but it is still waiting for the data. The cache line will be in the IS^{D}I (IM^{D}I) state if while waiting for its data in (i.e. in IS^{D} (IM^{D})), a store request to the same cache line is issued by another core.

C. Timer-related Coherence States

Figure 3 shows an illustrative example for the operation of PENDULUM. We use this example throughout the remainder of this paper to illustrate the operation of PENDULUM, the semantics of the coherence states, and the analysis of the coherence latency. Figure 4 shows the timer-related coherence states and their transitions in PENDULUM.

The following rules dictate the operation of timers and the corresponding PENDULUM states and transitions. (1) Once a core receives a cache line, it starts its corresponding timers (ST in Figure 4). This is the situation at timestamp ❶ in Figure 3 for Core0. In addition, if the cache line was in the IS^{D}I or IM^{D}I state, it moves to the S^{T}I (Figure 4a) or M^{T}I (Figure 4b) state, respectively, and waits for the timer to expire before it invalidates itself. For instance, in Figure 3, Core1 was at IM^{D} state at ❹ waiting for data of the cache line A. Then, it moves to IM^{D}I state at ❺ because Core2 issued its store request to A. Once Core1 receives its data at ❽, it moves to the M^{T}I state. (2) If a timer expires and the cache line is not requested by another core with the same criticality level as the timer, then the timer is replenished (RT). This is important to avoid unnecessary invalidations. (3) If a core has a cache line in S state and another core requests the same cache line to modify (OtherGetM), the current core moves to S^{T}I state, waits for its timer to expire (WT), and issues a self-invalidation to the arbiter (SelfInv). This is the transition from S to S^{T}I in Figure 4a. In Figure 3, Core3 moves to the S^{T}I state at ❹ because of Core1's getM request and waits for its timer expiry, which occurs at ❼. (4) If a core owns a cache line (M state) and another core issues a request to this cache line, a similar process to (3) occurs with the exception that the owner moves to M^{T}I state (Figure 4b). In Figure 3, Core0 moves the state of cache line A from M to M^{T}I at ⓭ because of Core3's request to A. (5) Multiple cores can share the same cache line in the S state without modification. In this case, every sharer has its own timer counting down based on when it obtained this cache line independent of the other cores. A requestor to that cache line has to wait for the core whose timer expires the last before it can obtain the cache line. In Figure 3 at the TDM slot between timestamps ❷ and ❸, both Core0 and Core3 have a



(a) $\mathsf{S}^T\mathsf{I}$: Transitioning from shared to invalid. (b) $\mathsf{M}^T\mathsf{I}$: Transitioning from modified to invalid. (c) $\mathsf{S}^T\mathsf{M}$: Transitioning from shared to modified.

Fig. 4: Timer-related states introduced by PENDULUM and their corresponding transitions.

TABLE III: PENDULUM private cache coherence states. Shaded rows are the timer-related states. WT: Wait for timer timeout, RT: Replenish timer, ST: Start timer. $msg/state$ denotes that a core Issues the message msg, and moves to coherence state $state$. Cells marked as "×" indicate that a particular transition cannot happen, and cells marked as "−" denote that a cache line in that state does not change state with a core event or bus event.

State	Core events				Bus events - common to Cr and nCr cores								
	Load	Store	Replacement	Timeout	OwnGetS	OwnGetM	OwnPutM	OwnSelfInv	AllInv	OwnSendData	Data	OtherGetS-Cr or -nCr	OtherGetM-Cr or -nCr
I	Issue GetS/IS^{AD}	Issue GetM/IM^{AD}	X	X	X	X	X	-	-	X	X	-	-
IS^{AD}	X	X	X	X	/IS^{D}	X	X	-	-	X	X	-	-
IS^{D}	X	X	X	X	X	X	X	-	-	X	load/S and ST	Criticality-dependent (Table IV)	
IS^{D}I	X	X	X	X	X	X	X	-	-	X	load/S^{T}I and ST		
S	Hit	/S^{T}M and WT	Issue SelfInv/SI^{A}	/S and RT	X	X	X	X	X	X	X	-	Issue SelfInv/S^{T}I and WT
S^{T}I	Hit	S^{T}M and WT	Issue SelfInv/SI^{A}	/SI^{A}	X	X	X	X	X	X	X	-	Issue SelfInv
SI^{A}	Hit	Issue SelfInv and GetM/SM^{A}	Issue Self-Inv	X	X	X	X	/SI	X	X	X	-	Issue SelfInv
SI	Hit	Issue GetM/IM^{AD}	/I	X	X	X	X	X	/I	X	X	-	-
S^{T}M	Hit	X	X	Issue SelfInv and GetM /SM^{A}	X	X	X	X	X	X	X	-	Issue SelfInv
SM^{A}	Hit	X	X	X	X	/IM^{D}	X	/IM^{AD}	X	X	X	-	Issue SelfInv
IM^{AD}	X	X	X	X	X	/IM^{D}	X	-	-	X	X	-	-
IM^{D}	X	X	X	X	X	X	X	-	-	X	store/M and ST	*	*
IM^{D}I	X	X	X	X	X	X	X	-	-	X	store/M^{T}I and ST	*	*
M	Hit	Hit	Issue PutM/MI^{A}	/M and RT	X	X	X	X	X	X	X	Issue Send-Data / M^{T}I and WT	Issue SendData /M^{T}I and WT
M^{T}I	Hit	Hit	Issue PutM and SendData/MI^{A}	/MI^{A}	X	X	X	X	X	X	X	Issue Send-Data	Issue SendData
MI^{A}	Hit	Hit	Issue PutM	X	X	X	WB/I	X	X	send data to re-questor/I	X	Issue Send-Data	Issue SendData

valid read-only version of A. Note that $\mathsf{S}^T\mathsf{M}$ transient state is still a read-only shared state. (6) If a core has a store request to a cache line that it has in S state, it has to wait for its timer to expire and then issue this store request on the bus. This is necessary to maintain coherent data sharing since S state implies that other cores can also have this line in S, which will not be invalidated until their timer expiration. Therefore, as shown in Figure 4c, if a core has a store request to a cache line in the S or the $\mathsf{S}^T\mathsf{I}$ state, it moves to $\mathsf{S}^T\mathsf{M}$, waits for timer expiration, invalidates, and then issues the GetM to the arbiter. This is Core0's request situation in Figure 3 at ❶. (7) $\mathsf{S}^T\mathsf{I}$ is effectively a shared state; therefore, loads to a cache line in this state are cache hits. Similarly, $\mathsf{M}^T\mathsf{I}$ is a modified state, where loads or stores to a cache line in this state are cache hits. (8) If a cache line is in $\mathsf{S}^T\mathsf{I}$ or $\mathsf{M}^T\mathsf{I}$ and a Timeout occurs or a replacement (Rep in Figure 4) is requested by the core, this ine will eventually move to the I state. In case of $\mathsf{M}^T\mathsf{I}$, a write back to the shared memory is also needed (PutM and SendData messages in Figure 4b). Dotted lines in Figure 4 denote that the cache line may move to other transient states until it observes its coherence messages and then move to the indicated stable state. However, these transient states are not related to the timers, and we refer the reader to [36] for the complete transition table.

D. Handling Mixed Criticality

Deploying a criticality-aware arbiter to protect Cr requests from interference by assigning them higher priority is not enough for MCS if tasks are simultaneously sharing data. In such systems, requests to shared memory are not necessarily atomic, since the data transfer might not start in the same

TABLE IV: Different PENDULUM transitions based on criticality levels.

Criticality level of owner core	State	Requesting core is Cr		Requesting core is nCr	
		Load	Store	Load	Store
Cr	IS^D	-	Issue SelfInv /IS^DI	-	Issue SelfInv /IS^DI
	IS^DI	-	Issue SelfInv	-	Issue SelfInv
	IM^D	Issue SendData /IM^DI	Issue SendData /IM^DI	Issue SendData /IM^DI	Issue SendData /IM^DI
	IM^DI	Issue SendData	Issue SendData	Issue SendData	Issue SendData
nCr	IS^D	-	reIssue GetS /IS^{AD}	-	Issue SelfInv /IS^DI
	IS^DI	-	reIssue GetS /IS^{AD}	-	Issue SelfInv /IS^DI
	IM^D	reIssue GetM /IM^{AD}	reIssue GetM /IM^{AD}	Issue SendData /IM^DI	Issue SendData /IM^DI
	IM^DI	reIssue GetM /IM^{AD}	reIssue GetM /IM^{AD}	Issue SendData /IM^DI	Issue SendData /IM^DI

slot in which the request was issued. This might happen for instance because another core already has the requested data in its private cache (as Figure 3 illustrates). In MCS, the cache coherence protocol has to ensure data correctness, while determining actions based on the criticality of memory requests. Otherwise, a Cr request may have to wait indefinitely because of a nCr's request to the same cache line as explained in Section II-C3. Accordingly, some of PENDULUM state transitions are dependent on the criticality of the communicating cores. These transitions are indicated as Criticality-depedent in Table IV. It is important to observe that these transitions and their corresponding states cover the situation of pending requests where a core issued a request to a cache line, but is still waiting to get the data (this is the *owner* core), and in the mean time another core requests an access to the same cache line (this is the *requestor* core). The transitions in Table IV can be classified into the following two categories. 1) The owner's criticality is higher than or equal to the requestor's criticality. In this case, the owner gets the data first, performs its access, and then invalidates its cache line after its corresponding timer expires. If the owner's request was a store, the owner sends the data as well after the timer expiration to the requestor. This is the case in Table IV, where the owner is Cr regardless of the requestor's criticality, or both the owner and the requestor are nCr. For instance in Figure 3, Core1 issues a store request to A at ❹ and moves to the IM^D state, while waiting for data. In the mean time, Core2 has a store request to A at ❺. Since both Core1 and Core2 are Cr, Core1 moves to the IM^{D}I state and gets access to A first (at ❽), before it invalidates (at ❾) such that Core2 gains access to A at ❿. This means effectively, accesses to same cache line from cores of same criticality are served in their arrival order. 2) The owner's criticality is lower than the requestor's criticality. This is the case where the owner is a nCr core, while the requestor is Cr. In this case, the owner's request will be preempted and the core has to reissue its request. This preemption only affects the nCr's request latency and has no effect on the system state. This is because the request was still pending and data transfer did not yet start. On the other hand, this preemption is necessary to provide the Cr cores with the independence from the nCr cores behavior on shared data, and thus, ensure a bounded latency for requests from Cr cores. In Figure 3, Core3 issues a GetS to A and moves to IS^D at ⓭ waiting for Core0's timer expiration. In the mean time, Core2 issues a GetM to A at ⓮. Since Core2 is Cr, while Core3 is nCr, Core3's request to A is preempted, and it moves to the IS^{AD} state, which means it is waiting for its request to A to be reissued.

E. Hardware Overhead

Two timer fields are added to each cache line, which adds to the overhead of the cache area. In our current implementation, to reduce this overhead, we designate the values of these timers to be a multiple of the arbiter period. This alignment of timer values to arbiter periods allows us to reduce the timer bits to 4 bits for each timer. This reduces the overhead by 75% compared to state-of-the-art time-based coherence protocols, which use 32-bit timers per cache line [33], [37]. We find that allocating 4 bits per timer is sufficient since it allows timers to go up to 16 periods of the arbiter schedule. It is worth noting that this overhead is low considering the current cache size in COTS platforms. For instance, the Intel i7 Haswell processor has an L1 cache size of 32KB [38], and a tag array of size 2KB to store state bits and address information. PENDULUM adds 8 bits per cache line, which results in an additional overhead of 0.5KB (1.47% additional overhead).

F. Supporting Systems with More than Two Criticality Levels

Supporting systems with arbitrary number of criticality levels is beyond the scope of this paper and is left as a future extension. Nonetheless, such an extension is feasible without requiring either significant changes in the operation of PENDULUM or additional overheads. Two main components need to be changed to support an arbitrary number of criticality levels: timers and criticality-dependent coherence states.

1) The number of timer fields will remain the same (two per core). However, their semantics will change. Instead of depending on whether the owner/requestor is Cr or nCr, they should depend on whether the requestor has a lower criticality than the owner or not. Accordingly, the two timer fields of each core will have the value of Timer(HCr) and Timer(LCr). Timer(HCr) applies if the requestor's criticality is higher than the owner's criticality, while the Timer(LCr) applies otherwise. A core Core$i$ has to invalidate its cache line after Timer(HCr) (Timer(LCr)) if the requestor has higher (equal or lower) criticality than Core$i$.

2) Similarly, the semantics of the coherence states that are dependent on the criticality (those tabulated in Table IV), has to change to reflect the relative criticality level of the requestor as compared to the owner. To exemplify, assume that Core$i$

issued a request to cache line A and is waiting for data in state IS^D and another Core$j$ issues a store request to A. If Core$j$ has a lower criticality than Core$i$, then Core$i$ will gain access to A first. On the other hand, if Core$j$ has a higher priroity than Core$i$, Core$i$ request is preempted and it has to reissue its request to A.

VI. Timing Analysis

In this section, we derive the latency bounds for a memory request issued by a Cr core to the shared memory. Towards doing so, we define three latency components: access latency (Definition 1), arbitration latency (Definition 2), and coherence latency (Definition 3). Any request to the shared memory incurs one or more of these latency components.

Definition 1: **Access latency**, L^{acc}, is the time consumed by a request, while it is performing the access and transferring the requested data to the core.
Access latency accounts for the time needed to perform the memory operation once the request is granted access to the requested data, and hence, it is independent of the arbitration policy and does not include the effect of any interference from other requests. It includes the time required to transfer the requested data from the shared memory (or another core's private cache) to the private cache of the requesting core. We assume that L^{acc} is a fixed latency. 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 [23] and DRAMs [39].

Definition 2: **Arbitration latency**, L_r^{arb}, is the latency incurred by a request r due to the arbitration on the shared memory bus. It is measured from the timestamp when r is issued to the bus until it is granted access to the memory. L_r^{arb} accounts for the latency suffered by a request due to other requests that are scheduled by the arbiter before this request regardless of their requested cache lines.

Definition 3: **Coherence latency**, L_r^{coh}, is the additional latency incurred by a request r due to the coherence protocol rules that ensure data correctness in the cache hierarchy. It is due to requests from other cores to the same cache line of r.

For any memory request from a Cr core, to incur the WCL, it has to incur both WC arbitration latency, WCL^{arb}, and WC coherence latency WCL^{coh}, in addition to the access latency. Equation 2 calculates this total WCL. Accordingly, to bound the memory latency incurred by any request, it remains to calculate WCL^{arb} and WCL^{coh}. We derive WCL^{arb} in Lemma 1 and derive WCL^{coh} in Lemma 2.

$$WCL = WCL^{arb} + WCL^{coh} + L^{acc} \tag{2}$$

Lemma 1: (***WC Arbitration Latency***) The WC arbitration latency of a request generated by any Cr core is:

$$WCL^{arb} = N_{Cr} \times SW$$

Proof: The proof directly follows from the arbitration policy. Since Cr cores are arbitrated using TDM and nCr cores are granted access only in slack slots, a request incurs the WC arbitration latency when it arrives such that it just misses its own core's slot; thus, it has to wait for a full TDM period. Since each Cr core has one slot per TDM period and the slot width is SW, the TDM period is $N_{Cr} \times SW$. ■

Lemma 2: (***WC Coherence Latency When Data Sharing Is Enabled Among All Cores***) The WC coherence latency, WCL^{coh}, incurred by a request issued by any Cr core is:

$$\begin{aligned} WCL^{coh} &= \mathsf{Timer(Cr, Cr)} + \mathsf{Timer(nCr, Cr)} - SW \\ &\quad + (N_{Cr} - 1) \times \big(\mathsf{Timer(Cr, Cr)} + (N_{Cr} - 1) \times SW\big) \end{aligned}$$

Proof: Let the Cr core under analysis to be Core$i$, which issues a request to a cache line A.
(1) Coherence interference from Cr cores: The WC coherence interference for Core$i$ occurs when all other Cr cores have pending store/write requests to the same cache line, A. Further, all these pending requests from other Cr cores are scheduled before Core$i$'s request. Since the coherence protocol enforces the SWMR invariant, only one core can obtain A to write at a time. Consequently, all these pending write requests to A are serialized. Each of these $N_{Cr} - 1$ other Cr cores obtains A in M state for a period of Timer(Cr,Cr). Moreover, since each core gains access to the bus only in its dedicated slot, once a core's timer expires, it can take a maximum of $N_{Cr} - 1$ slots before another core obtains A. This results in a total coherence interference of $(N_{Cr}-1)\times\big(\mathsf{Timer(Cr, Cr)}+(N_{Cr}-1)\times SW\big)$ from other Cr cores.
(2) Coherence interference from nCr cores: In addition, Core$i$ can suffer a maximum coherence delay of $\mathsf{Timer(nCr, Cr)}-SW$ from nCr cores, due to the fixed-priority scheduling. This maximum delay occurs when a nCr core gains access to the same cache line requested by Core$i$ just before Core$i$ sends its request to the arbiter. This nCr core needs one slack slot to gain such access, which explains the subtracted SW term.
(3) Coherence interference because of a write to a non-modified cache line: Finally if Core$i$'s request to A was a write, it can suffer additional latency if it already has A in S state in its private cache. This is because as explained in Section V, PENDULUM disallows write hits to non-modified cache lines. Therefore, a write request to a block in S state has to wait for the timer to expire before it is sent to the arbiter. Since a Cr core keeps a cache line with another Cr pending request for a maximum of Timer(Cr,Cr), the WC of this additional latency if Core$i$'s request to A was a write request and Core$i$ has A in S state is Timer(Cr,Cr).

From 1–3, WCL^{coh} when data sharing is enabled among all cores is as calculated in Lemma 2. ■
Figure 3 shows this WCL^{coh} for a system with 3 Cr cores and one or more of nCr cores (the number of nCr is irrelevant). Core0 is the core under analysis, which has a cache line A in S state. At timestamp ❶, Core0 has a store request to A. However, this request has to wait until the expiration of A's timer (timestamp ❸) before it can be sent to the arbiter. In the mean time, and directly before the expiration of A's timer in Core0's private cache (timestamp ❷), Core3, which is a

nCr core obtains a slack slot and issues a read request to A. Since the arbiter has no pending Cr requests to A (Core0's store request is still pending for the timer expiration and not send to the arbiter yet), Core3 gains access to A. At timestamp ❸, Core0's timer expires; however, Core0 still has to wait for its own slot before the arbiter issues its store request to A. At timestamp ❹, Core1 utilizes its own slot to send a store request to A; however, it has to wait for Core3's Timer(nCr,Cr) timer to expire. Similar situation occurs for Core2 in its slot at timestamp ❺. In this case Core2 has to wait for both Core3 and Core1. Core0's store request to A is issued at timestamp ❻, but it has to wait for Core3, Core1, and Core2 accesses to A. Although Core3's timer expires at ❼, Core1 waits for its slot at ❽ to gain access to A. After the expiration of Core1's timer at ❾, Core2 obtains A at ❿ and his timer expires at ⓫. Finally, Core0 obtains A and perform its store operation at ⓬. From Figure 3, the total latency that Core0's store request to A suffers is $\mathsf{Timer}(\mathsf{Cr},\mathsf{Cr}) + \mathsf{Timer}(\mathsf{nCr},\mathsf{Cr}) - SW + 2\times\big(\mathsf{Timer}(\mathsf{Cr},\mathsf{Cr})+2\times SW\big)+3\times SW$. The last term $(3\times SW)$ is the arbitration and access latency.

Lemma 3: (***WC Coherence Latency When Data Sharing Is Enabled Among Critical Cores Only***) The WC coherence latency, WCL^{coh}, incurred by a request issued by any Cr core if data is only shared among Cr cores can be calculated as:

$$WCL^{coh} = \mathsf{Timer}(\mathsf{Cr},\mathsf{Cr}) \\ + (N_{Cr}-1)\times\big(\mathsf{Timer}(\mathsf{Cr},\mathsf{Cr}) + (N_{Cr}-1)\times SW\big)$$

Proof: If the system disallows data sharing between Cr and nCr cores, a Cr core will not suffer coherence interference from the nCr. The proof directly follows from Lemma 2 by eliminating the coherence interference delays from the nCr. ■

VII. Correctness of PENDULUM

In order to ensure the correctness of PENDULUM under all possible states and transitions, we use a combination of three methodologies. All of them exhaustively cover all protocol states, which provides evidence of the correctness of the proposed protocol. 1) **Manual Testing**. We handcraft synthetic benchmarks that are carefully designed to cover all possible state transitions, including both stable and transient states. We then execute these benchmarks and log detailed debugging messages that capture the behavior of PENDULUM in every cycle. Finally, we examine these logged messages to ensure the correctness of PENDULUM. 2) **Random Tests**. We utilize the Ruby Random Tester that is included with the gem5 simulator [40] to generate 10 million random test requests to the shared memory hierarchy with PENDULUM as the implemented coherence protocol. Then similar to the manual testing, we examine all detailed logs to ensure correctness for all tests. 3) **Data Correctness**. We prototype PENDULUM in the gem5 micro-architectural simulator [40], and use the SPLASH-2 benchmarks as a representation of real benchmarks with shared data [41]. SPLASH-2 benchmarks comprise of data verification routines that report the correctness of the output produced by the multi-threaded benchmark routines. We found that all SPLASH-2 benchmarks on PENDULUM execute correctly, and the verification routines validate the data output for the multi-threaded benchmark routines.

VIII. Evaluation

Setup. We prototype PENDULUM in gem5 [40], a micro-architectural simulator that models the memory subsystem and coherence protocol with high precision. Table V tabulates the considered system parameters. We set accesses to the LLC to be cache hits to eliminate the interference on off-chip main memory and focus solely on the latency overhead of maintaining coherence copies of shared data in the cache hierarchy. DRAM interference is outside the scope of this work and existing works can be used to predictably manage accesses to main-memory (e.g. [42], [43]) or bound its interference (e.g. [39]). These works are orthogonal to PENDULUM and their delays are additive to those derived in this work [44].

Benchmarks. In our evaluation, we use both real benchmarks and synthetic benchmarks. For the real benchmarks, we use SPLASH-2 [41], a multi-threaded benchmark suite (Section VIII-A). The purpose of the synthetic benchmarks is to stress the behavior of the evaluated solutions (Section VIII-B).

A. SPLASH-2 Parallel Benchmarks

We use the SPLASH-2 benchmarks to compare PENDULUM with prior works both from performance and WCL aspects.

1) Average-Case Performance: Figure 5 compares the performance of PENDULUM with the MSI conventional (non-predictable) coherence protocol as well as the following three predictable approaches: Uncache shared, Task mapping, and PMSI. The details of these approaches are discussed in Section II. For PENDULUM, we set the timer values to be one TDM period for each of them. The effect of various timer values is then studied in Section VIII-C. Each of the benchmarks in Figure 5 consists of four parallel threads that share data among each other. To emulate the data sharing among Cr and nCr, two of the threads are mapped to two Cr cores, and the other two are mapped to two nCr cores. For PMSI, all cores are considered critical since it does not distinguish between different criticalities and equally allocates service to all cores. All results in Figure 5 are normalized to the MSI execution time. Results show the advantage of utilizing cache coherence in increasing system performance. PENDULUM introduces 42% slowdown on average compared to conventional MSI protocol. This is up to $1.9\times$ better performance than Task mapping (66% on average) and up to $3.48\times$ better performance than Uncache shared (78% on average). This is because cache coherence allows for simultaneous sharing of data, which as opposed to task mapping technique, does not limit system parallelism and as opposed to Uncache shared, allows for private cache hits for shared data. Additionally, PENDULUM achieves up to 13% and 4% on average better performance than PMSI.

2) Worst-Case Per-Request Latency: Figure 6 depicts the per-request WCL for SPLASH-2 benchmarks using different approaches. Uncache all does not use private caches at all for

TABLE V: Evaluation setup.

Param	Configuration
System	A quad-core system with 2 Cr and 2 nCr cores
Core	Scalar, in-order pipeline, 2GHz operating frequency
Cache	L1-D, L1-I = 16KB direct-mapped, 64B cache line, 3 cycle latency shared L2 (LLC), 1MB, 8-way set associative
Arbiter	Cr cores get higher priority by assigning them dedicated TDM slots, where nCr gain access on slack slots only. Slot width is 50 cycle

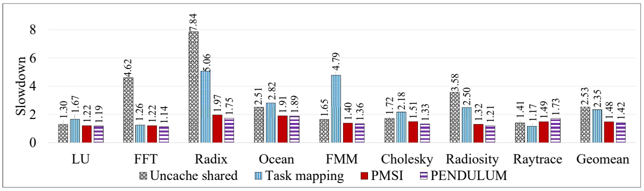


Fig. 5: Performance slowdown compared to MSI protocol.

applications with shared data. We make two main observations from Figure 6. 1) PENDULUM achieves much better per-request WCL compared to PMSI. The main reason is that PENDULUM is a criticality-aware coherence protocol that carefully accounts for the core criticality in its coherence transitions as we detailed in Section V-D. Therefore, it limits the coherence interference from nCr cores. 2) The WCL of Uncache all, Uncache shared, and Task mapping is lower than PENDULUM. The reason is that these solutions avoid the coherence latency (and thus, reduces the per-request WCL) through disallowing simultaneous access to shared data as discussed in Section II which comes at the expense of increasing the task's overall execution time from other aspects. While Uncache all and Uncache shared increase the number of cache misses by not caching shared data, Task mapping limits task parallelism. Therefore, the values of those approaches' WCL in Figure 6 are not representative sense they do not take into account the stated factors.

B. Synthetic Benchmarks

We design seven synthetic benchmarks to stress the behavior of PENDULUM and highlight interesting observations. The first benchmark is the Synth-All, which is designed to stress maximum data sharing across cores. In this benchmark all cores (Cr and nCr) execute the same sequence of memory operations on the same shared data. In addition to Synth-All, we design the six workloads tabulated in Table VI with varying cache locality to highlight the role of timers. We introduce the metric *request interval* to vary the cache locality in the synthetic benchmarks. The request interval defines the time interval between memory requests to the same cache line by a core. As Table VI shows, we vary the request interval in these benchmarks for both Cr and nCr cores such that they exhibit a varying cache locality behavior. This stresses the timers behavior since the timer values determine the period a cache line resides in the private caches of the cores. Since PMSI is the most relevant to this work as it enables

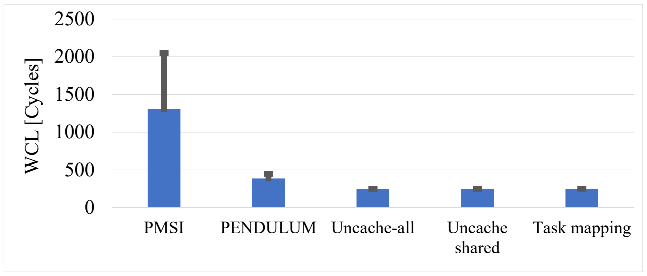


Fig. 6: Per-request analytical (T-sharp bar) and observed/experimental WCL (colored bar) for SPLASH-2.

TABLE VI: Synthetic workloads.

Workload	Request interval	
	(Cr)	(nCr)
Synth-A	10 cycles	20 cycles
Synth-B	10 cycles	10 cycles
Synth-C	10 cycles	5 cycles
Synth-D	20 cycles	10 cycles
Synth-E	5 cycles	10 cycles
Synth-F	5 cycles	5 cycles

predictable and simultaneous data sharing, we focus these synthetic experiments on comparing PENDULUM with PMSI. Similar to Section VIII-A, because PMSI does not support mixed criticality and provide equal service to all cores, all cores are considered critical for PMSI. On the other hand, for PENDULUM, we assign two cores as Cr and the other two as nCr. Additionally, we compare two possible configurations of a MCS: 1) a system where data is shared only among cores of same criticality (Shared Intra-Criticality Only in Figures 7 and 8), and 2) a system where data is shared across criticality (i.e. among both Cr and nCr cores), which is shown as Shared both Inter and Intra-Criticality in Figures 7 and 8.

1) Average-Case Performance: Figure 7 delineates the speedup that PENDULUM achieves over PMSI for both data sharing with only same criticality and data sharing across criticalities with an average speedup of 20% and 33%, respectively. These results align with the discussion in Section II. PENDULUM achieves this speedup by allowing cores to retain their ownership of cache lines in their private caches for a dedicated time, which is determined by the configurations of the timers. On the other hand, PMSI follows the conventional snooping methodology, where cores have to invalidate their cache lines immediately once requested to be modified by another core, which creates the ping-pong effect we discussed in Section II-C2. Figure 7 also shows that PENDULUM in a system that restricts data sharing among only cores of same criticality achieves better performance for most of the synthetic benchmarks. This is intuitive since such system has less interference among cores (thus, less invalidations). However, this comes at the expense of constraining data flow among tasks in the system. Whether data is shared among same criticality or across criticalities is use-case dependent and PENDULUM is compatible with both system configurations.

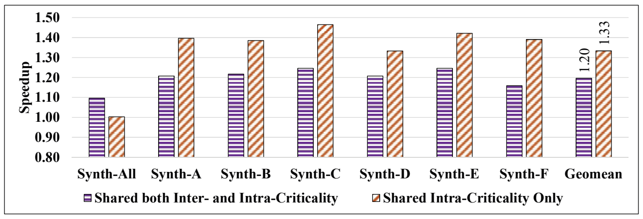


Fig. 7: PENDULUM performance with both data sharing among 1) cores of same criticality and, 2) all cores. All values are normalized to PMSI performance.

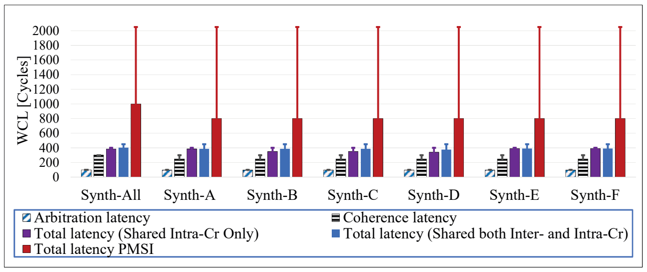
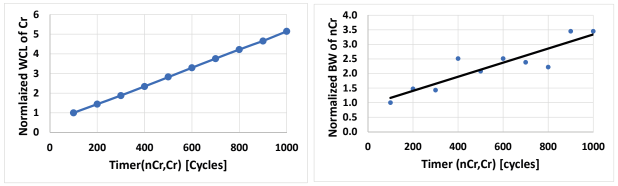


Fig. 8: Analytical (T-sharp bars) and experimental (colored bars) WCL for the benchmarks in Table VI.

2) *Worst-Case Per-Request Latency:* Figure 8 illustrates the WCL of PENDULUM for both data sharing only among same criticality cores and among all cores as well as the WCL of PMSI. It also shows the worst-case values for the latency components: arbitration, and coherence (access latency to shared memory has a fixed value of 50 cycles). The solid bars in Figure 8 are the observed WCL from experiments, while the T-shape above each bar represents the analytical WCL from Section VI. As Figure 8 highlights, PENDULUM achieves much lower WCL than PMSI. PENDULUM achieves $3.56\times$ less analytical latency than PMSI when the system shared data among all cores and observed WCL shows a similar trend. Another observation from Figure 8 is that the gap between analytical and observed WCLs for PENDULUM is much smaller compared to PMSI, which indicates that PENDULUM also achieves higher predictability with tighter bounds. This can be explained based on the discussion in Section II-C3. PMSI is not designed for MCS and hence it falls short of meeting MCS requirements. PENDULUM, on the other hand, is criticality-aware and is able to minimize the interference that a Cr core suffers from nCr cores. This is achieved by both the fixed-priority arbitration that dedicates slots to Cr only and service nCr at best effort utilizing slack slots, and the criticality-aware coherence protocol with different behavior based on cores' criticalities.

C. Role of Timer Configuration

In Section V-A, we discussed in details the effect of the timer values of PENDULUM in the WCL and average performance of both Cr and nCr cores. We conduct a set of experiments, where we vary each of the four timers and study its effect in a quad-core system comprising two Cr and two nCr



(a) WCL of Cr cores. (b) Bandwidth of nCr cores.

Fig. 9: Impact of Timer(nCr,Cr).

cores. Because of the space limitations, we only show in this section the results for the Timer(nCr,Cr). Timer(nCr,Cr) is the most interesting timer since it rules the relationship between Cr and nCr cores and its value addresses the trade-off between WCL of Cr and average performance of nCr, which are the two commonly considered objectives in MCS. Figure 9 delineates the findings of this study. We increase Timer(nCr,Cr) value in multiples of 100 cycles since it has to be a multiple of TDM periods as explained in Section V-A. We normalize all results to the value of $\text{Timer}(\text{nCr}, \text{Cr}) = 100$ cycles. As expected, increasing Timer(nCr,Cr) increases the WCL of Cr cores. This aligns with the analytical bound in Lemma 2. On the other hand, by increasing Timer(nCr,Cr), nCr cores achieve higher bandwidth since their hit rate increases by locking cache lines in their private caches for at least Timer(nCr,Cr).

IX. Conclusion

Enabling data sharing is an important feature for MCS for emerging domains including automotive and avionics. However, this should be done without imposing new restrictions in the system or applications that hinder applicability. Towards this target, we propose PENDULUM a solution to enable simultaneous data sharing for MCS tasks without imposing any restrictions on the system schedulability or the tasks' legacy software. PENDULUM is a cache coherence protocol designed for MCS with configurable capabilities that enable the MCS designer to address the trade-offs between predictability and average performance requirements from both critical and non-critical tasks. Comparisons with state-of-the-art solutions show that PENDULUM achieves flexibility and better performance, while ensuring predictability.

X. Acknowledgements

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

References

[1] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakraborty, "Time-triggered implementations of mixed-criticality automotive software," DATE '12, EDA Consortium, 2012.

[2] G. Xie, G. Zeng, Z. Li, R. Li, and K. Li, "Adaptive dynamic scheduling on multifunctional mixed-criticality automotive cyber-physical systems," *IEEE Transactions on Vehicular Technology*, 2017.

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

[4] M. G. Hill and T. W. Lake, "Non-interference analysis for mixed criticality code in avionics systems," in *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, IEEE, 2000.

[5] K. K. Venkatasubramanian, S. Nabar, S. K. Gupta, and R. Poovendran, "Cyber physical security solutions for pervasive health monitoring systems," in *E-Healthcare Systems and Wireless Communications: Current and Future Challenges*, IGI Global, 2012.

[6] C. Kotronis, G. Minou, G. Dimitrakopoulos, M. Nikolaidou, D. Anagnostopoulos, A. Amira, F. Bensaali, H. Baali, and H. Djelouat, "Managing criticalities of e-health iot systems," in *17th International Conference on Ubiquitous Wireless Broadband (ICUWB)*, IEEE, 2017.

[7] B. Lesage, I. Puaut, and A. Seznec, "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)*, ACM, 2012.

[8] N. C. Kumar, S. Vyas, R. K. Cytron, C. D. Gill, J. Zambreno, and P. H. Jones, "Cache design for mixed criticality real-time systems," in *International Conference on Computer Design (ICCD)*, 2014.

[9] 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)*, IEEE, 2016.

[10] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst, "A mixed critical memory controller using bank privatization and fixed priority scheduling," in *International Conference on Embedded and Real-Time Computing Systems and Applications*, IEEE, 2014.

[11] J. Jalle, E. Quiones, J. Abella, L. Fossati, M. Zulianello, and F. J. Cazorla, "A Dual-Criticality Memory Controller (DCmc): Proposal and Evaluation of a Space Case Study," in *IEEE Real-Time Systems Symposium*, IEEE, 2014.

[12] 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 *IEEE Real-Time Systems Symposium (RTSS)*, IEEE, 2016.

[13] M. Hassan, A. M. Kaushik, and H. Patel, "Predictable Cache Coherence for Multi-core Real-Time Systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, 2017.

[14] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A Survey on Cache Management Mechanisms for Real-Time Embedded Systems," *ACM Computing Surveys*, 2015.

[15] D. Hardy, T. Piquet, and I. Puaut, "Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches," in *IEEE Real-Time Systems Symposium (RTSS)*, IEEE, 2009.

[16] B. Lesage, D. Hardy, and I. Puaut, "Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures.," in *International Conference on Real-Time and Network Systems*, 2010.

[17] 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 Systems Review - Special Topics*, 2015.

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

[19] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware Support for WCET Analysis of Hard Real-time Multicore Systems," in *ACM Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[20] F. Hebbache, M. Jan, F. Brandner, and L. Pautet, "Shedding the Shackles of Time-Division Multiplexing," in *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

[21] V. Suhendra and T. Mitra, "Exploring Locking & Partitioning for Predictable Shared Caches on Multi-cores," in *ACM Annual Design Automation Conference (DAC)*, 2008.

[22] M. Schoeberl, W. Puffitsch, and B. Huber, "Towards time-predictable data caches for chip-multiprocessors," in *Springer International Workshop on Software Technolgies for Embedded and Ubiquitous Systems (IFIP)*, 2009.

[23] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Making Shared Caches More Predictable on Multicore Platforms," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[24] D. Guo, M. Hassan, R. Pellizzoni, and H. Patel, "A Comparative Study of Predictable DRAM Controllers," *ACM Transactions on Embedded Computing Systems (TECS)*, 2018.

[25] P. Stenstrom, "A survey of cache coherence schemes for multiprocessors," *IEEE Computer*, 1990.

[26] M. M. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Communications of ACM*, 2012.

[27] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, 2011.

[28] S. L. Min and J.-L. Baer, "Design and analysis of a scalable cache coherence scheme based on clocks and timestamps," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 1992.

[29] X. Yuan, R. Melhem, and R. Gupta, "A timestamp-based selective invalidation scheme for multiprocessor cache coherence," in *IEEE Workshop on Challenges for Parallel Processing (ICPP)*, 1996.

[30] M. Lis, K. S. Shim, M. H. Cho, and S. Devadas, "Memory coherence in the age of multicores," in *IEEE International Conference on Computer Design (ICCD)*, 2011.

[31] K. S. Shim, M. H. Cho, M. Lis, O. Khan, and S. Devadas, "Library cache coherence," 2011.

[32] X. Yu and S. Devadas, "Tardis: Time traveling coherence algorithm for distributed shared memory," in *IEEE International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[33] Y. Yao, G. Wang, Z. Ge, T. Mitra, W. Chen, and N. Zhang, "Efficient timestamp-based cache coherence protocol for many-core architectures," in *ACM International Conference on Supercomputing (ICS)*, 2016.

[34] IEC61508, "Functional safety of electrical/electronic/programmable electronic safety-related systems," 2010.

[35] A. Alhammad and R. Pellizzoni, "Trading cores for memory bandwidth in real-time systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[36] N. Sritharan, A. M. Kaushik, M. Hassan, and H. Patel, "PENDULUM: A Cache Coherence Protocol for Mixed Criticality Systems, A Technical Report."

[37] I. Singh, A. Shriraman, W. W. Fung, M. O'Connor, and T. M. Aamodt, "Cache coherence for GPU architectures," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[38] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual," 2016.

[39] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in COTS-based multi-core systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[40] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, *et al.*, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, 2011.

[41] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ACM Annual International Symposium on Computer Architecture (ISCA)*, 1995.

[42] M. Hassan, H. Patel, and R. Pellizzoni, "PMC: A requirement-aware DRAM controller for multicore mixed criticality systems," *ACM Trans. Embed. Comput. Syst.*, 2017.

[43] Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst Case Analysis of DRAM Latency in Multi-requestor Systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[44] H. Yun, R. Pellizzon, and P. K. Valsan, "Parallelism-Aware Memory Interference Delay Analysis for COTS Multicore Systems," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.