Predictably and Efficiently Integrating COTS Cache Coherence in Real-Time Systems

Mohamed Hossam
McMaster University, Hamilton, Canada

Mohamed Hassan
McMaster University, Hamilton, Canada

Abstract

The adoption of multi-core platforms in embedded real-time systems mandates predictable system components. Such components must guarantee the satisfaction of the timing constraints of various applications running on the system. One of the components that can break the system predictability is cache coherence, which ensures the correctness of shared data. This paper proposes a solution towards the enablement of predictable cache coherent real-time systems. The solution uses existing COTS coherence protocols and proposes a methodology to integrate them with legacy real-time arbiters without imposing any required modification to either of them. Doing so, the paper also works as an exploratory study of the integration of various coherence protocols with various predictable arbitration schemes leading to a total of 12 different architecture configurations. Evaluation against four state-of-the-art predictable coherence solutions as well as COTS-based solutions show that the proposed approach achieves the tightest existing latency bounds among predictable solutions with minimal performance degradation over the COTS ones.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Real-time system architecture; Computer systems organization → Multicore architectures

Keywords and phrases Coherence, Shared Data, Caches, Multi-Core, Real-Time, Memory

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.17

Supplementary Material PCC-sim, an open source cache simulator
Software: `https://gitlab.com/FanosLab/pcc-sim`

1 Introduction

Compared to traditional real-time embedded systems, modern applications of embedded systems impose fundamentally new set of challenges. Examples of these applications are prevalent in domains such as automotive, unmanned air vehicles (UAVs), space, and industry 4.0. With the transition towards more autonomy in these domains, the challenges are correlated to processing massive amounts of data, which requires unprecedented computation power as well as memory bandwidth. Moreover, this data processing must happen as quick as its incoming rate from the external physical environment (e.g. camera frame rate in a self-deriving car or an incoming vital signal from a patient). This dictates a minimal acceptable average performance of the computing system. Meanwhile, the fact that these are safety-critical systems, they have the stringent requirement of predicable performance. This is expressed in terms of deadlines that should never be exceeded under all conditions. To be able to provide this predictable performance, the hardware architecture itself should be predictable to enable the derivation of reasonable bounds on the worst-case execution time (WCET) of all running tasks.

Architecting computing systems to meet all the aforementioned requirements becomes a challenging task since they can conflict with each other. An example of such conflict that is relevant to the focus of this paper is data sharing. Most existing research in the predictable management of multi-core hardware assumes tasks do not share data [8]. This is because data sharing can infringe the isolation properties promised by these techniques [18]. Nonetheless, this is no longer possible for the aforementioned domains since inter-task communication through data sharing is a key to the functionality of these systems [28, 17]. For example, in automotive, measured values from several sensors need to be consumed by multiple functions [11].

Considering average performance requirements, cache coherence seems to be the appealing solution to enable data sharing both from academic [22, 23] and industry [9] perspective. Therefore, this paper focuses on cache coherence as the data sharing mean in multi-core real-time systems. Despite having a recent attention from the real-time community, the approach followed by the majority of existing works is to ensure predictability by mandating modifications to either the commercial-off-the-shelf (COTS) coherence protocols only [14], the legacy predictable interconnects or arbitration schemes [17], or both [15, 18, 27, 20, 19]. **In contrast, this paper contributes to the efforts of enabling predictable and coherent sharing of data in real-time systems by proposing PCC: a solution that integrates COTS coherence protocols into legacy predictable real-time arbitration schemes without requiring any modifications to either of them.** This solution is based on the following observation. The usually abstracted bus architecture in the real-time research is physically composed of several parallel buses in COTS platforms. In particular, the interconnect between private caches and the shared cache either deploys a communication bus for the communication among the private caches, which is separate from the bus connecting these private caches to the shared cache [3], or it enables several point-to-point connectivity that allows for overlapping transfers [1, 7]. *This enables the data to be sent to different destinations simultaneously; in particular, from a core's private cache to another private cache as well as to the shared memory.* A more thorough discussion about this observation is presented in the system model in Section 4. Leveraging this observation, the paper makes the following contributions.

1) It illustrates how to predictably integrate COTS coherence protocols into the legacy predictable real-time arbiters without imposing any architectural modifications to the protocol itself nor to the underlying predictable arbitration scheme. This is key since it has been established by prior works that directly doing so will lead to unpredictable behaviors [18]. However, we show how exploiting the architectural capability mentioned in the previous observation can restore predictability to the real-time multi-core system upon integrating cache coherence to it.

2) The predictability of the solution is proven by a formal timing analysis that we introduce in Section 6. Unlike existing works, this analysis is generalized to apply to various real-time arbiters as well as various COTS coherence protocols. Additionally, a key aspect of this work is that the provided bounds stand the same regardless of the pipeline architecture of the cores, whether in-order or out-of-order (OoO).

3) To confirm the claimed integrability, we deploy a wide set of COTS coherence protocols as well as predictable arbitration schemes. In addition to the modified-shared-invalid ($\mathcal{MSI}$) protocol that is adopted by most existing works, we also fully implement the $\mathcal{MESI}$ (E refers to Exclusive) and $\mathcal{MOESI}$ (O refers to Owned) protocols, which, unlike the simple $\mathcal{MSI}$ protocol, are common on multi-core platforms. For instance, the $\mathcal{MESI}$ protocol is adopted by the ARM's most-recent Cortex-R82 [4], while the $\mathcal{MOESI}$ is adopted by ARM's A53 processor [2]. For predictable arbiters, we exemplify the generality of the solution by implementing time division multiplexing (TDM), round robin (RR), weighted RR (WRR), and harmonic RR (HRR). This results in 12 different implemented and studied cache coherent architectures.

4) These rich system configurations enable us to conduct extensive case studies, which in turn lead to several novel observations about the various design trade-offs of choosing the coherence protocol as well as the arbitration mechanism from the predictability perspective of multi-core real-time systems. These observations are discussed in details in Section 7.
5) We compare against four predictable cache coherent techniques [15, 18, 19] as well as against conventional COTS coherence techniques. Results show that PCC is able to achieve the tightest bound for existing predictable coherence solutions, with a minimal performance degradation compared to COTS solutions.

2 Background

In this section, we cover the fundamentals of cache coherence protocols and shared bus arbitration.

2.1 Coherence Protocols

A Coherence protocol is the mechanism that cache controllers employ in multi-core systems to ensure the correctness of the data. The correctness is achieved by guaranteeing that all the cores have access to the latest version of the data. Thus, coherence protocols enforce Single-writer-multiple-reader (SWMR) invariant to keep the coherency of the data. The basic protocol that many COTS architecture implements is $\mathcal{MSI}$.

$\mathcal{MSI}$ protocol consists of three main states: Modified(M), Shared(S), and Invalid(I) where each cache line in the private cache should be either in one of these states or in the transition to one of them. M state grants read and write permissions to the core that has the cache line. Due to SWMR invariant, only a single core can have a certain cache line in M state at a time, and other cores cannot privately cache this line during this time. Cores can request lines for modification by issuing GetM message on the shared bus. On the other hand, S state is a read only state, where multiple cores can have the same cache line in this state. Cores request lines for read by broadcasting GetS message on the bus. The last state is I which indicates that the data of a cache line is not available in the private cache or the data is stale. I state does not allow reading or writing to the data.

Extensions can be applied to $\mathcal{MSI}$ by adding one or more states such as the exclusive (E) and the owner (O) state. These extensions result in the famous protocols: $\mathcal{MESI}$, $\mathcal{MOSI}$, and $\mathcal{MOESI}$. E state is similar to S state as both are read only states, but E state indicates that only one core has this cache line in its private cache. This allows such core to move from E to M silently without issuing GetM message. The other state, O is also a read only state, but it gives the core the ownership of the cache line, meaning this core should respond to other cores' requests for this line instead of the shared memory.

Transient States. Besides these standard states, which are called stable states, there are number of states that indicate the transitions between the stable states, and they are called transient states. Transient states are crucial due to the non-atomicity of the interconnect between cache memories. For instance, if core C_i requires to write to a cache line in the I state, it will issue a GetM message and wait for receiving the data. Before the data is received, the cache line can neither be in the I nor the M states; therefore, a transient state is required to define this transitional period. Conventionally, this state is named IM^{d}, which shows the source and destination states, and the superscript indicates the reason of the state ($^{\mathsf{d}}$ indicates waiting for data). In order to complete this example, we can assume that while C_i is waiting for data, it observes a GetM message from another core C_j. Accordingly, C_i should change the state of the line from IM^{d} to $\mathsf{IM}^{\mathsf{d}}\mathsf{I}$. $\mathsf{IM}^{\mathsf{d}}\mathsf{I}$ indicates that after the data arrives and write is performed, the cache line should be moved to I state. Some of these transient states are depicted in Figure 2 and discussed within the example in Section 3.

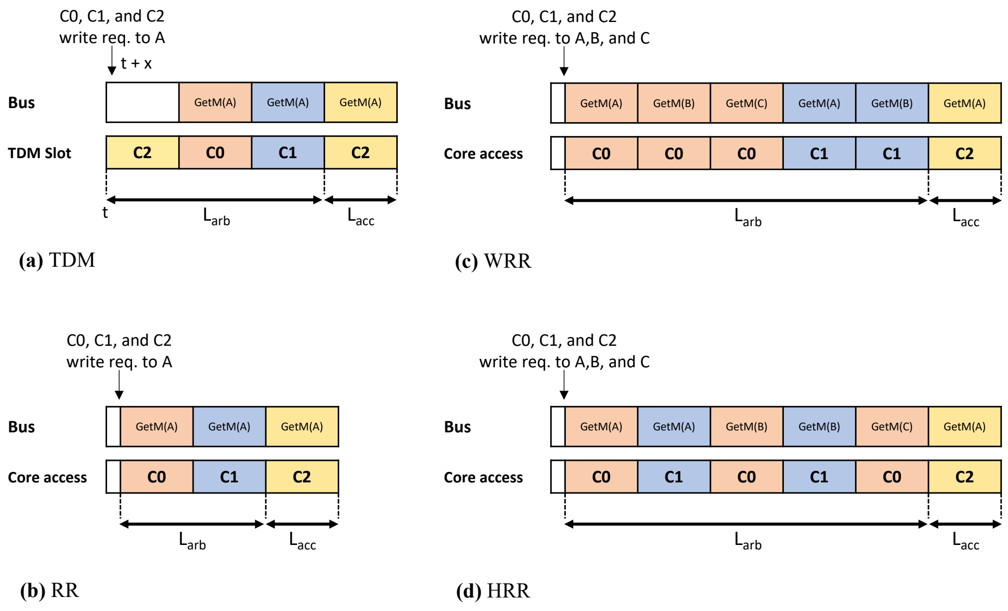

Figure 1 The worst-case latency of different bus arbiters. C_2 is the core under analysis.

2.2 Bus Arbitration

In embedded systems, the communication medium between private caches from one side and the shared memory from the other side is usually a shared bus. A bus arbiter is responsible for organizing accesses of the cores to the bus. Figure 1 shows the worst-case latency (WCL) of different predictable arbiters. A predictable arbiter should grant access to the bus to the requesting core in a predictable bounded latency. Figure 1a shows the latency of TDM arbiter, which dedicates a fixed time slot for each core to access the bus. The WCL of TDM is govern by $WCL = N \times S$, where N is the number of cores and S is the slot width. The width of the slot is chosen to have $S \geq (L_{arb} + L_{acc})$, where L_{arb} is the bus latency to broadcast a request and L_{acc} is the latency of data transfers over the bus. Similarly, Figure 1b shows the latency of RR arbiter, which is a dynamic arbiter that keeps a cyclic list of cores with ready requests. WCL of RR is calculated by $WCL = (N-1) \times (L_{arb} + L_{acc})$. There are multiple variants of RR, the most common are WRR and HRR. Both WRR and HRR allow cores to have different arbitration weights. WRR's WCL can be calculate as shown in Figure 1c, for a core C_j WCL is $\sum_{i=0,i\neq j}^{N-1} W_i \times (L_{arb} + L_{acc})$, where W_i is the weight of core C_i. WCL of HRR is calculated differently, as shown in Figure 1d the weights of the cores are distributed harmonically, therefore the WCL of C_j in HRR can be calculated using $(\lceil HP/W_j \rceil - 1) \times (L_{arb} + L_{acc})$, where HP is the hyper-period of the cyclic list of the cores, and it is equal to the summation of all the weights.

3 Related Work and Motivation

In the way of adopting the multi-core systems in real-time applications, many efforts have been conducted to facilitate this adoption by predictably managing interference among different cores upon accessing shared hardware resources in the system. Examples of these resources are caches [5, 12, 14, 15, 18, 19, 21, 27], interconnects [17, 20], and main memory [10, 13, 16]. Among these works, the most relevant to this paper are the ones focusing on enabling coherent data sharing through hardware coherence protocols [14, 15, 18, 19] and the ones aiming at predictably arbitrating accesses to the shared cache [17, 20].

PMSI [15] provides predictable cache coherence by modifying the legacy $\mathcal{MSI}$ protocol and implementing it on top of a unified TDM bus. The work was extended in [18] by adopting a $\mathcal{MESI}$-based solution. PMSI (and PMESI) suffers from two main drawbacks. 1) Its WCL is quadratic in the number of cores, which makes it very pessimistic. 2) It requires modifications to both the COTS coherence protocol and the underlying architecture. The first drawback hinders its usability for real-time systems with tight latency requirements, while the second one makes it hard to adopt by industrial entities since designing and verifying new coherence protocols is known to be one of the most tedious architectural tasks [23, 24].

In order to elaborate PMSI's operation graphically, we use the example in Figure 2. The example crafts a scenario that highlights the key features and drawbacks of PMSI. Moreover, it is used to explain the rest of the related approaches, and in Section 5 we utilize the example to present PCC's operation and how it tackles the other approaches' downsides. The example shows the different latency components of the write request from core C_2 to the memory location A. Initially, at ❶ cores C_0 and C_2 have write requests to A which is cached in the shared memory. At ❷, C_1 has a read request from the memory location B which is modified by C_2. Afterwards at ❼, C_2 has a read request from A followed by a read request from B. Towards the end at ❽, C_1 requests another read from B.

The breakdown of PMSI's latency is illustrated in Figure 2a, where cores issue their request messages only during their dedicated TDM slots. C_0 issues a GetM message at ❶ and receives the data from the shared memory in the same slot. On the other hand, C_2 issues its message at ❸ which entails that C_0 changes A's state from M to MI$^{\mathsf{wb}}$ according to PMSI protocol. MI$^{\mathsf{wb}}$ indicates a transition from M state to I state after writing data back to the shared memory. Regarding the request latency, its first component is L_{arb}, which is the time between a core having a request until the request is broadcast on the bus. Since PMSI does not allow cache-to-cache transfers, C_2 has to wait for a complete TDM period to receive A after C_0 writes it back to the shared memory at ❹. The duration between the request broadcast and the data being ready to be sent (the time between ❹ to ❺) is the inter-coherence latency ($L_{inter-coh}$), which appears because of the coherence interference between the cores, such as the write-back from C_0. Nonetheless, C_2 cannot receive A at ❺ because of the previous request from C_1 at ❷ which requires C_2 to write B back to the shared memory. Accordingly, C_2 receives A at ❻ instead of ❺, and this delay is defined as the intra-coherence latency ($L_{intra-coh}$) that results from the intra-core interference between a core's demanding requests and its write-backs. The last latency component is L_{acc}, and the total latency per request is the summation of all the aforementioned components. Clearly, the WCL of such behavior is very pessimistic as the latency of c_2's write(A) illustrates. Finally, the requests at ❼ and ❽ are all hits, because PMSI allows read from the modified cached lines (such as A) as well as previously modified lines (such as B). These two features, which are inherited from $\mathcal{MSI}$, result in a relatively good average-case performance as shown in Table 1.

DISCO [14] makes the observation that write requests are the reason of PMSI's excessive WCL due to the need for write-backs. Hence, it proposes to tighten this WCL by discriminating between read and write requests. This is done by prohibiting modified cache lines from being stored in the private L1 caches of cores; instead, all write requests must be serviced at the shared memory directly. The tight WCL can be observed from Figure 2b, where the write requests from C_0 and C_2 are serviced during the same slots they are issued in at ❶ and ❸. Moreover, C_1's request at ❷ is serviced directly from the shared memory, unlike the case of PMSI, because C_2 is not allowed to privately cache the modified B line. Nonetheless, WCL comes at the cost of average performance which is embodied in the misses of the read

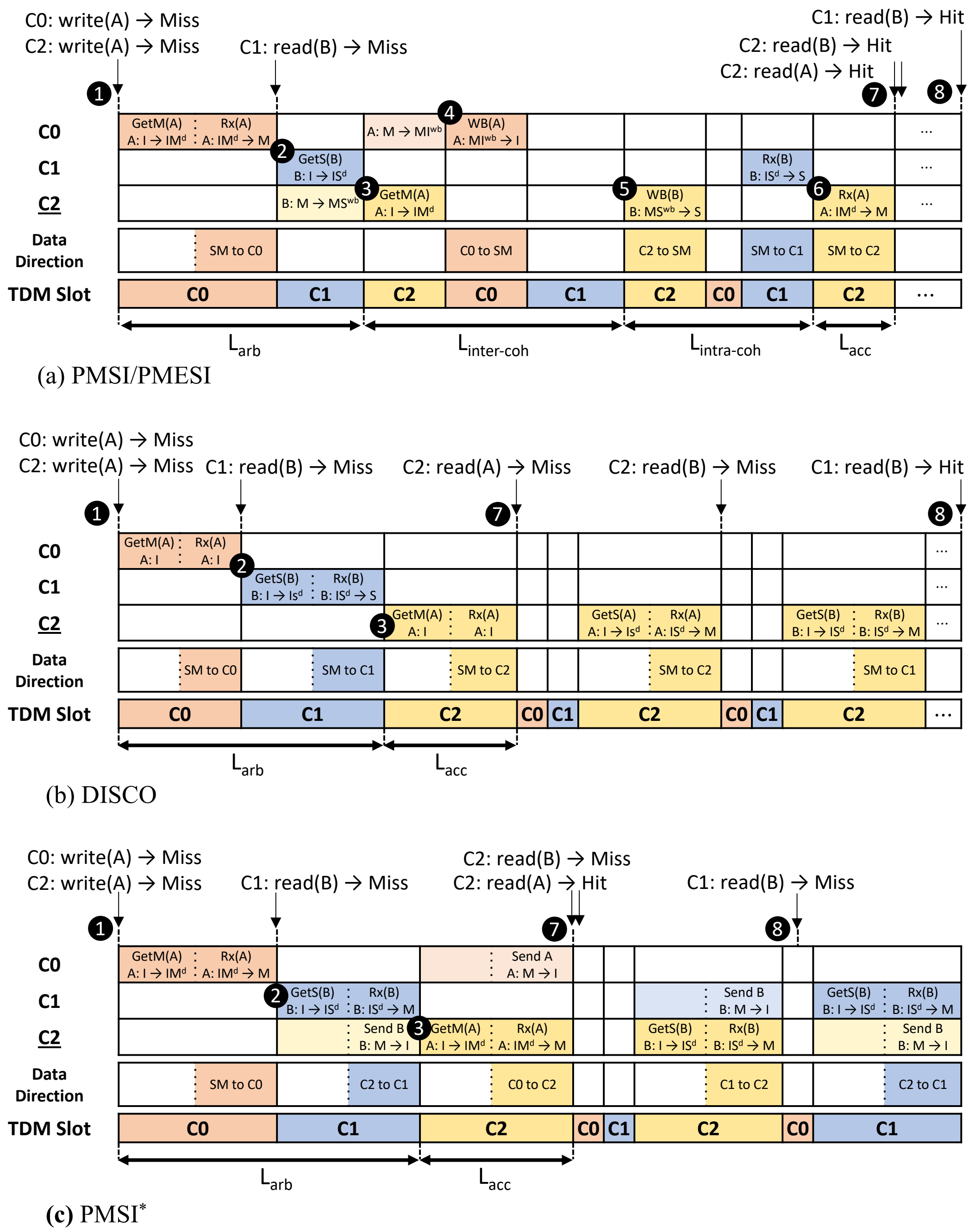

Figure 2 The memory latency of the write request to A from core C_2 in a 3-core system that implements unified bus to connect privates cache with the shared memory. The bus uses TDM arbitration with fixed slot width; however, the figure shows different sizes for the slots in order to fit into the page width. In the beginning of the scenario, A resided in the shared memory, while B was modified by C_2.

Table 1 Summary of the properties of the related work along with PCC's properties. ✗ indicates that the ✗ is not supported, while ✓ indicates it is supported. features with several marks indicate a score for each work. For example, ✗✗✓✓ indicates a score of 2/4, ✗✗✗✓ indicates 1/4, ✗✓ indicates a score of 1/2, and ✓✓ indicates a score of 2/2.

Related work	Per-req. WCL	Bus Arch.	Supports OoO Exe.	Supports COTS Protocols	Supports different arbiters	Supports C2C Data Transfer	average performance
PMSI/ PMESI	Quadratic	Unified	✗✗	✗	✗	✗	✗✗✓✓
DISCO/ DISCO$_{SharedW}$	Linear	Unified	✓✓	✗	✓	✗	✗✓✓✓
PMSI*/ PMESI*	Linear	Unified	✗✗	✗	✗	✓	✗✗✗✓
PISCOT	Linear	Split	✗✓	✓	✗	✓	✓✓✓✓
PCC	Linear	Unified	✓✓	✓	✓	✓	✓✓✓✓

requests from C_2 that come after ❼. It also requires hardware support to enable selective bypassing of L1 caches for write requests. The author also proposed DISCO-SharedW in [14] as an enhanced version allowing the caching of modified cache lines in the L1 caches if they are to a private data that is not shared amongst cores. This optimization provides a good balance between a tight WCL and good performance. However, it assumes the availability of a prior information about the shared data. Accordingly, DISCO-SharedW scores a relatively high average-case performance in Table 1's comparison.

The PMSI* and PMESI* solutions are proposed in [19] to be modified versions of the original PMSI/PMESI protocol. Similar to DISCO, they aim at reducing the quadratic WCL of PMSI by mitigating the impact of write-backs. Unlike DISCO, they limit the number of write-backs that have to go to the shared memory. This is achieved by deploying two techniques. 1) They enable direct communication between private caches, similar to this work. 2) They remove some of the standard transitions in the $\mathcal{MSI}$ and the $\mathcal{MESI}$ protocols. In particular, if a core owns a line in the modified state, they disallow such line from being shared among several cores afterwards; i.e., such line can only be accessed by a single core at a time. The reason for this modification is to make this line directly transferable among cores while avoiding the need to transfer this line to the shared memory (until eviction). An example to this transition is C_1's read request at ❷, where C_1 receives B directly from C_2 but C_2 invalidates B after its transfer. While achieving a linear WCL in number of cores, this solution suffers from a significant performance penalty due to disabling simultaneous sharing of such lines and the possibly ping-pong effect as a result. The ping-pong effect appears between the read requests from C_2 and C_1 at ❼ and ❽; thus, PMSI* and PMESI* are given the worst performance score in Table 1.

PISCOT [17] follows a different approach by deploying a split-bus interconnect that implements a TDM request bus to broadcast coherence requests and a first-come first-serve (FCFS) response bus to transfer data responses. PISCOT is the only previous solution that enables the deployment of conventional coherence protocols without modifications. It also leverages the split-bus interconnect to achieve high average performance, while maintaining predictability with tight latency bounds. Another aspect of PISCOT compared to all existing works is its support to OoO cores by considering cores with multiple outstanding requests. Nonetheless, we find this support to be limited since it only services one request at a time; mainly to ensure the tight latency bounds. PCC similar to PISCOT enables the adoption of COTS coherence protocols in real-time systems without modifications. In doing so, unlike PISCOT, which requires a specific split-bus architecture, PCC enables the usage of legacy prevalent real-time bus architectures and arbiters. PCC also supports OoO cores and since it allows all requests to be non-preemptively serviced once their corresponding messages are broadcasted, it does not put any constraint on the OoO behavior of cores, while offering the tight WCL that is independent of the core pipeline architecture.

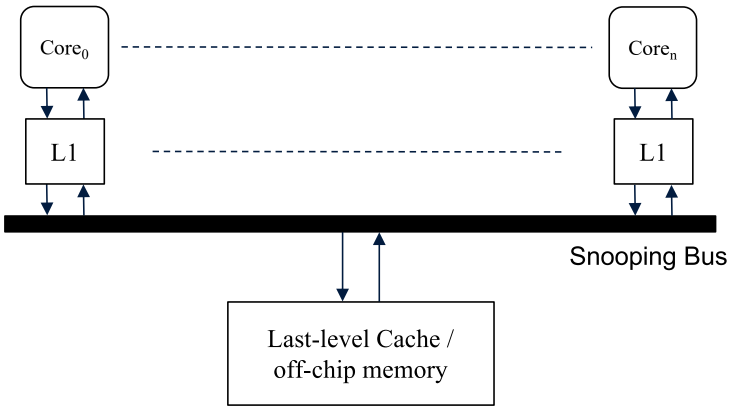

Figure 3 The system model.

Table 1 summarize the differences between all the aforementioned approaches, and from that we can conclude the features that should be present in the proposed approach. Initially, the per-request WCL should be linear with the number of cores without compromising the average-case performance. Also, PCC should be independent of the coherence protocol therefore it can incorporate COTS protocols without any modifications. Moreover, it uses unified bus architecture with the ability of cache-to-cache data transfer to guarantee both tight WCL bounds and high performance. Finally, it should support OoO pipelines without changing the WCL, and it should not assume any prior information about shared data. All the previous point are also summarized in the last row of Table 1.

4 System Model

We consider a multi-core system that has N cores as shown in Figure 3.

Cores Architecture and Cache Hierarchy. Unlike most of the existing related work, we do not put a constraint on the pipeline architecture of the cores, they can be in-order or OoO or a mix of both. Each core has a private cache (L1). In addition, the system contains a shared memory, which it can be a last level cache, off-chip memory, or both of them. The writes from L1 to the shared memory is handled using write-allocate write-back policy, and the cache hierarchy is inclusive such that data existing in any L1 is a subset of the shared memory's data. It is important to highlight that the proposed solution works for general cache architectures. For example, the solution works seemingly for different number of cache levels, where each core has several private caches and then the last private level of all cores connect to a shared cache. In that case, the solution works for the bus connecting to the shared cache. An exhaustive enumeration of all possible cache architecture is beyond the scope of this paper; thus, for conciseness, we focus on the model depicted in Figure 3. Another important notice is that we assume that the data will always be serviced within the depicted memory hierarchy in Figure 3 (i.e. the shared cache will always have the line in the valid state). This assumption is only to avoid the other sources of interference that are beyond the scope of this paper (e.g. I/O interference [6] or off-chip memory interference [10]).

Cache Coherence. The data is kept coherent among the private caches by incorporating any of the standard COTS coherence protocols. In this work, we exemplify by adopting three protocols: $\mathcal{MSI}$, $\mathcal{MESI}$, and $\mathcal{MOESI}$.

Bus Architecture. L1 caches and the shared memory communicate through a *logical* shared snooping bus. We use the term logical here to differentiate between the logical bus model and the actual *physical* implementation details of this bus. As far as the paper is concerned, we make few assumptions about (requirements) the logical bus to be able to derive the bounds. Aside from these requirements, the *physical* implementation of the bus can be realized using any technique. These assumptions are as follows.

1) The bus allows the direct data transfers between L1 caches. Moreover, a transfer from a sender L1 cache to another can be overlapped with a transfer to the shared memory. In other words, a data sent from a core can be received by the shared memory and another core simultaneously. We find these assumptions to reflect techniques adopted in COTS architectures. For example, different existing ARM processors allow for direct transfers between private L1s, this includes processors both real-time (e.g. Cortex-R82 [4]) and application (e.g. Cortex-A53 [2]) families. Additionally, the data-coherent bus connecting private L1 caches and the snooping control unit (SCU) in Arm's MPCore processor is separate from the bus connecting the SCU to the shared L2 cache, where the latter is AXI-based (e.g. in A9) or with the Cache Coherence Interconnect extensions to the AXI interface (CCI-400) such as in A15. On the other hand, for the CCI-550, the data interconnect is mentioned to be a fully connected cross bar [1]. Another vendor's example is the QorIQ processor family from NXP, where the CoreNet Coherency Fabric enables point-to-point connections to pipeline the transfers between cores and shared memories [7]. Either having a dedicated bus or several parallel point-to-point connections will enable the required overlapped transfer.

2) The bus has a logical unified architecture. This means transferring a coherence message and a data message cannot be overlapped. During anytime instance, either a coherence message is being broadcast or a data message is being transferred. It is important to note that this assumption does not prevent the coherence and data messages from being sent on two different realized physical buses. It only requires unifying their arbitration such that their transfers are not overlapped. This is key to enable the integration of COTS coherence in predictable arbitration schemes with tight latency bounds, especially for OoO cores as we will discuss in section 6.

3) Cores are granted access to the bus using a predictable arbitration scheme. This work is not limited to a specific arbiter type, and we provide results, in Section 7, for TDM, RR, WRR, and HRR arbiters. Once a request is granted access to the bus by the arbiter, it will remain in service and no other request will be granted access until the in-service request is fulfilled. The maximum time to service any request is assumed to be L_{acc}. For slot-based arbitration schemes (such as TDM), the time slot of the bus arbiter should be long enough to fit the latency of broadcasting coherence messages besides transferring data (i.e. slot width should be at least L_{acc}).

The data sharing model. we don't assume any constraints on data sharing or the shared address space. Additionally, our proposal can work with the timing interference management solutions like memory partitioning and coloring. Also, we don't assume any restrictions on the task scheduler, so any task can run on any core without any implications on the system predictability.

5 Proposed Solution

PCC leverages the observations about the potential architectural features deployed by COTS platforms that are specified in the system model (Section 4) to facilitate the predictable integration of cache coherence in real-time systems without drastic degradation of performance.

In particular, the operation of PCC makes use of these two features: 1) direct cache-to-cache communication, and 2) a data transfer from a core can be simultaneously sent to another core as well as the shared memory. As a result, PCC operates according to Theorem 1.

▶ **Theorem 1.** *Once a request, $R_i(A)$ to any cache line A from any core C_i is granted access to the shared bus under PCC, it will be non-preemptively serviced without interference from any other request.*

Proof. Under the PCC system model described in Section 4, once a request from any core C_i is granted access to the shared bus, the actions conducted towards its fulfillment falls under one of the following three scenarios (depending on the type of the request as well as the specifications of the adopted coherence protocol).

1. **The requested cache line is already owned by C_i** In that case, to fulfil the request, it needs only to broadcast a coherence message. This will be the only action needed for requests that do not necessitate a data transfer. An example is the transition from O state to M state as a result of a write request under $\mathcal{MOESI}$ protocol.
2. **The requested cache line is owned by the shared memory.** In that case, C_i will also need to broadcast a coherence message and then receive the data from the shared memory. This will occur if the data is owned by the shared memory.
3. **The requested cache line is owned by another core, say C_j** In that case, C_i needs to broadcast a coherence message and then receive the data from C_j.

For Scenario 1, it is clear that once $R_i(A)$ is granted access to the bus using any predictable arbiter, it is an exclusive access, where it can issue its message non-preemptively; and hence, finishes its service.

For Scenario 2, after $R_i(A)$ gains access to the bus and broadcast its message (similar to scenario 1), it also requires to receive the requested data from the shared memory. According to the system model in Section 4, this also happens directly once $R_i(A)$'s message is broadcasted without any interruption from other requests.

For Scenario 3, there are two sub-scenarios as follows. Scenario 3(a) is the case where the owner core C_j needs to send the data to C_i only and not the shared memory. This is the case for example if $R_i(A)$ is a getM(A) request, while A is modified in C_j's private cache. This scenario is similar to Scenario 2 since it requires a single data transfer, while the source of the data is now a core and not the shared memory. Therefore, same argument applies similar to Scenario 2 such that $R_i(A)$ finishes without any interruption.

Scenario 3(b) is a bit more involved as it requires two data transfers and not one. In this scenario, C_j needs to send the data to both C_j and the shared memory. This occurs for instance if $R_i(A)$ is a getS(A) request, while A is modified in C_j's private cache. Leveraging the observation about the bus architecture in Section 4, PCC is able to conduct these two transfers in parallel; and hence, enables $R_i(A)$ to also finish in this case without any interruption. ◀

5.1 Illustrative Example

Now, we show the operation of PCC in action by applying it to the same example in Figure 2 and discuss how PCC offers predictability with tight bounds and no required modifications to the coherence protocol.

Figure 4 shows the latencies of PCC in a system that utilizes TDM arbiter and $\mathcal{MSI}$ coherence protocol. At ❶, C_0 and C_2 have write requests to A, and each request is broadcast, afterwards, to the bus at the beginning of its core's TDM slot. At ❸, C_2, the core under analysis, is granted access to the bus and it issues its request. During the same slot of C_2,

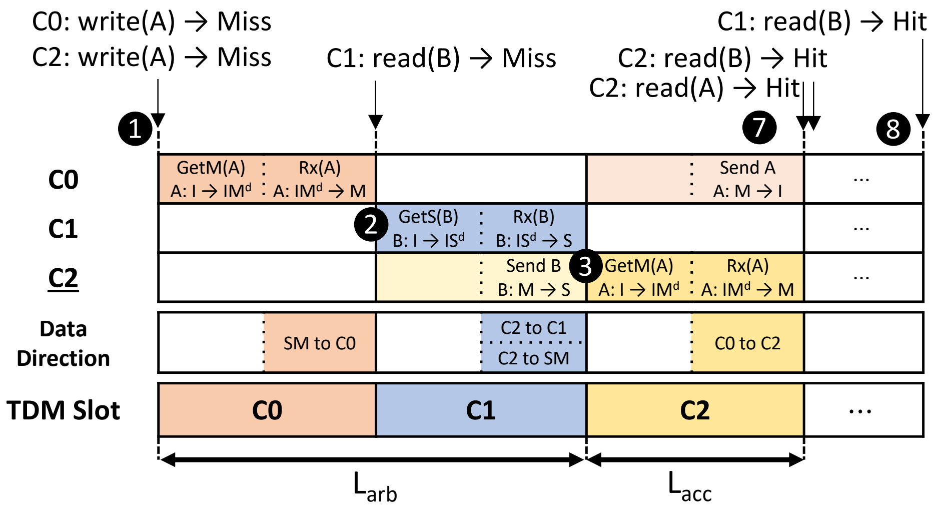

Figure 4 The memory latency of a write request to A from core C_2 in a 3-core system that uses PCC along with TDM arbiter and $\mathcal{MSI}$ coherence protocol. This figure follows the same scenario, initial conditions, and the numbering as in Figure 2.

C_0, the previous owner of A, responds with the data. Accordingly, this example contains only two type of latencies L_{arb} and L_{acc}, which they are the only types that a single request can incur in case of PCC . Unlike PMSI, PCC does not suffer from $L_{inter-coh}$ or $L_{intra-coh}$ because of the fact that requests are served in a single access slot to the bus (Theorem 1). Additionally, a core cannot have a pending write-back and a request during its access slot because the write-backs are handled during the requestor's access slot, and core's access slots are for its demanding requests only.

PCC leverages COTS coherence protocols which ensures high average-case performance. This can be observed from the example in two instances. First at ❷, where C_1 requests reading B which is owned by C_2. Consequently, C_2 responds by sending B to C_1 as well as to the shared memory; hence, both C_1 and C_2 are allowed to keep B in S state, unlike PMSI*. The second instance at ❸, after C_2 completes its write to A, PCC allows C_2 to keep A in M state, unlike DISCO. This results in access hits for the read request to A at ❼ and the read requests to B at ❼ and ❽.

6 Timing Analysis and Predictability Guarantees

In this section, we show that PCC solution satisfies the predictability invariants proposed in [15]. Besides, we derive PCC's per-request WCL and the total worst-case memory latency incurred by a task.

6.1 Satisfying Predictability Invariants

According to [15], a system that utilizes COTS coherence protocol along with a predictable arbiter cannot guarantee a predictable memory latency. Hassan et al., also, provided in [15], 6 invariants to test the predictability of cache memory systems. Accordingly, we show in this section that PCC satisfies all the invariants, which means COTS protocols with any predictable arbiter can be predictable if cache-to-cache data transfers are allowed according to the described system model.

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.

▶ **Lemma 2.** PCC *satisfies Invariant 1.*

Proof. Allowing a core to issue a request on the bus is the bus arbiter's responsibility, and according to the system model defined in Section 4, PCC utilizes predictable arbiters. Predictable arbiters allow cores to issue requests only in their own dedicated access slots. ◀

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

▶ **Lemma 3.** PCC *satisfies Invariant 2.*

Proof. Let two requests, $R_i(A)$ and $R_j(A)$ from two different cores Ci and Cj, respectively. Both requests target the same cache line which is owned by the shared memory. If $R_i(A)$ and $R_j(A)$ appear on the bus at t_i and t_j, respectively, where $t_i < t_j$, then according to Theorem 1, $R_i(A)$ once its message is broadcast at t_i, it will not be interrupted by any other request including $R_j(A)$ until it is serviced, which implies here that the shared memory finishing sending the data to Core Ci. Therefore the shared memory services the request according to the order of their appearance on the bus which is the same order of their arrival to the shared memory. ◀

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

▶ **Lemma 4.** PCC *satisfies Invariant 3.*

Proof. Let two requests, $R_i(A)$ and $R_j(B)$ from two different cores Ci and Cj, respectively. The requests target different cache lines which are both owned by Ck. Similar to the shared memory, if R_i appears first on the bus, C_k has to respond with the data to Ci before any other request can broadcast its message (Theorem 1). In conclusion, cores respond immediately to requests once they appear on the bus, therefore the requests' arrival order is respected. ◀

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

▶ **Lemma 5.** PCC *satisfies Invariant 4.*

Proof. The coherence protocols dictate how the cache controllers deal with the writes to the non-modified lines. According to the system model in Section 4, PCC incorporates COTS protocols that necessitate a modification request (GetM or UpgM) to be broadcast before writing to a non-modified line (i.e., a line in S state). (1)
According to Lemma 2, cores issue requests only during their access slot. (2)
From (1) and (2), a write request to a non-modified line in the core's private cache should wait for the core's access slot to the bus. ◀

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

▶ **Lemma 6.** PCC *satisfies Invariant 5.*

Proof. Assume that Ci owns non-modified cache line A in its private cache, and at time t_i, it has a write request, $R_i(A)$, to A. In addition, assume that Cj requested access (read or write), and its request $R_j(A)$, to A is broadcast on the bus at time t_j, where $t_j < t_i$. Invariant 5 breaks if PCC allows serving $R_i(A)$ before $R_j(A)$. (1)
Whereas, Lemma 5 enforces Ci to wait until it is granted access to the bus before proceeding with $R_i(A)$. Let this to happen at time $t_{i+\delta}$, where by construction $t_{i+\delta} > t_i$ (2)
Finally, Theorem 1 dictates that once a request is broadcast, it will be serviced before any other request can be broadcast or serviced (3)
From (1) – (3) and since $t_j < t_{i+\delta}$, it necessitates that $R_j(A)$ will be serviced before $R_i(A)$; thus, PCC satisfies Invariant 5. ◀

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

▶ **Lemma 7.** PCC *satisfies Invariant 6.*

Proof. From Theorem 1, cores will never have pending responses to others' requests during their own access time. The reason for that is because of the principle of non-preemptively servicing the requests once they are granted access to the bus. Accordingly, the arbitration between a core's own generated requests and its responses always chooses the requests due to the absence of pending responses. Therefore, PCC has the effect of this arbitration layer intrinsically without implementing it. ◀

6.2 Per-Request WCL Analysis

According to the example in Section 5, it is clear that the latency of a single request depends mainly on the type of the bus arbiter.

▶ **Lemma 8.** *The per-request latency is calculated as in Equation 1, where WCL_{arb} is the worst-case arbitration latency.*

$$WCL_{perReq} = WCL_{arb} + L_{acc} \tag{1}$$

Proof. In worst-case, such request has to wait for WCL_{arb} before it can be granted access to the bus by the arbiter. Once it is granted access to the bus, according to Theorem 1, a core's request is non-preemptively serviced. Based on the system model, this service consumes a maximum latency of L_{acc}. Therefore, the per-request worst-case latency is as depicted in Equation 1. ◀

▶ **Lemma 9.** *The processing latency of any request under PCC is calculated as in Equation 1 regardless of the pipeline architecture of the cores in the system (whether in-order, out-of-order or a combination of both).*

Proof. For an in-order core, the proof is straightforward. An in-order core can have a maximum of one request in-flight at any given time. Therefore, such request do not suffer any queuing delay from requests of the same core (it is always the head of the queue). Such request suffers a worst-case processing latency as proven by Lemma 8. Additionally, such core causes a maximum interference of one request on other cores since it cannot have several requests simultaneously requiring service per construction.

An out-of-order core, in contrast, can have several simultaneously outstanding requests. We now prove that this behavior does not impact the processing latency of requests from such core, nor impacts other cores. First, for requests from the core itself, we are bounding the processing latency of any request, which is the latency suffered by such request once it becomes at the head of the queue of its corresponding core. This is because this processing latency is the considered one to be used when calculating the overall's task WCET []. As a result, no queuing delay needs to be added as a component to the latency in Lemma 8. Second, for the interference impact from this core on other cores, we prove that it still adheres to Lemma 8 as follows. 1) Under any of the predictable arbiters considered in the system model, each core gets a guaranteed turn to access the bus regardless of the behavior of other cores or the number of their outstanding requests. As a result, the arbitration latency component, WCL_{arb} remains the same. 2) By construction of PCC and as proven by Theorem 1, once a request is granted access to the bus, it entertains a non-preemptive service until it is fulfilled. This is regardless of the behavior of other cores. As a result, the access latency component, L_{acc} remains the same. From 1) and 2), we finish the proof for the out-of-order core case. ◀

6.3 Total Task's Worst-Case Memory Latency Analysis

A task's WCET can be computed as follows: $WCET = WCCT + WCML$, where the $WCCT$ is the worst-case computation time of the task executing on the core, and the $WCML$ is the total worst-case memory latency suffered by the task upon accessing the memory. We now show how to compute the $WCML$ using the per-request WCL derived in Section 6.2.

WCL can be simply calculated using the per-request WCL calculated in Equation 2 as follows, where R_T is the total number of memory request issued by the task under analysis.

$$WCML = R_T \times WCL_{perReq} \tag{2}$$

Effect of Dirty Line Replacements. Another latency component that should be considered is the effect of the write back requests due to L1 cache evictions or replacement In worst-case, every request can trigger an eviction of a dirty cache line; and hence, creates a write back request that it has to wait for. As a result, The WCML in Equation 2 changes to the value calculated in Equation 3. This is because every request now has to wait for an additional write back request that also susceptible to WCL_{perReq} latency in worst case.

$$WCML = 2 \cdot R_T \times WCL_{perReq} \tag{3}$$

In some systems, the number of dirty line evictions can be constrained to the number of write requests. Accordingly, calculating WCML can be carried out using Equation 4, where R_W is the number of write requests.

$$WCML = R_T \times WCL_{perReq} + R_W \times WCL_{perReq} \tag{4}$$

This is the case for instance for the $\mathcal{MSI}$ protocol. However, COTS protocols that implement E state (e.g. $\mathcal{MESI}$ or $\mathcal{MOESI}$) require PutM message for the lines in the E state, which are clean (non-modified) lines. As a result, using Equation 4 with such protocols is not sufficient and can lead to unsafe bounds. In that case, using a more conservative bound such as the one in Equation 3 is the safe decision.

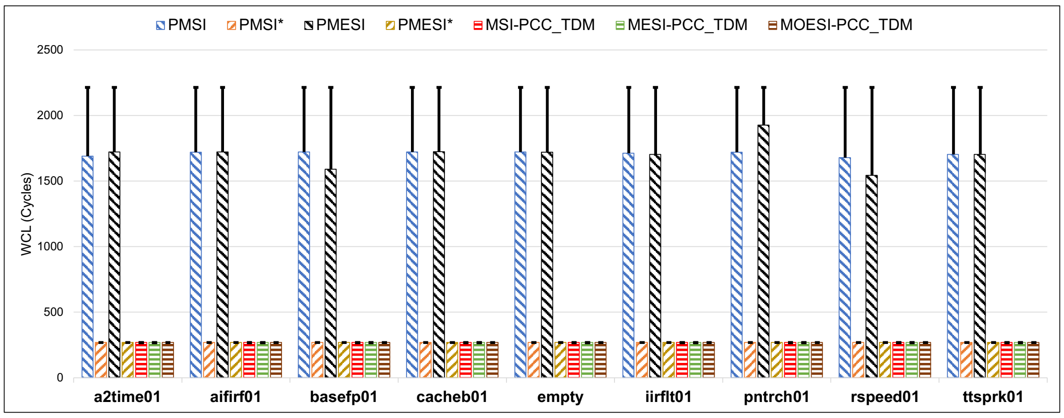

Figure 5 Per-request WCL of running EEMBC benchmarks, where T-bars represent the analytical value and solid bars are for the observed WCL among all the requests.

Effect of Hit/Miss Cache Analysis in the Presence of Coherence. Equation 3 is safe and sufficient to be used in the case of lack of more available information from the task analysis; however, WCML can be tighten using information from running the task under analysis in isolation (i.e., run the task on a core while turning off the other cores). For instance, the assumption that all demanding requests from a task incur WCL_{perReq} is rather pessimistic, since in most typical cases number of L1 hits is higher than the misses, and the hit latency (L_{Hit}) is much smaller than the WCL_{perReq}. However, it is not feasible to infer the number of hits from the isolation analysis due to the absence of coherence interference which decreases the hit rate. Moreover, the effect of coherence interference is not limited only to the shared data, but it can affect the hit rate of private data as well unless the target system offers data isolation between private and shared data in L1. Therefore, systems without data isolation should adhere to $WCML$ calculated by Equation 3. Whereas, if the system is capable of isolating shared data from private data (for example by providing a separate cache partition for each, $WCML$ can be more tightened by Equation 5. Equation 5 is based on the fact that if private and shared data are isolated from each other, they will not be able to evict each other, and an isolation analysis is performed to calculate the number of requests for private data that hit in L1 ($R^{iso}_{privHit}$), the number requests for private data that miss in L1 ($R^{iso}_{privMiss}$), the number of write-backs due to replacements to private data (R^{iso}_{Repl}), and the number requests for shared data (R_{Shared}). Since no assumption can be made about requests to shared data, all of them are assumed to be misses, and further suffering from write-back replacement delays.

$$WCML = \big(R^{iso}_{privHit} \times L_{Hit} + (R^{iso}_{privMiss} + R^{iso}_{Repl}) \times WCL_{perReq}\big) + \big(2 \times R_{Shared} \times WCL_{perReq}\big) \quad (5)$$

7 Evaluation

In order to evaluate our proposed solution, we performed a number of experiments to compare between PCC and state-of-the-art solutions. Additionally, we conducted other experiments to explore the effects of different combinations of bus arbiters with coherence protocols along with different execution modes (in-order and OoO). Throughout the experiments, we used a

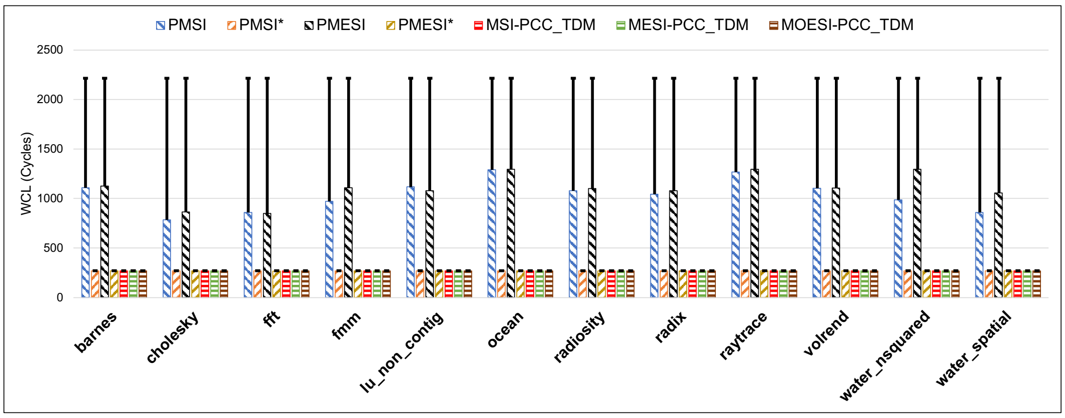

Figure 6 Per-request WCL of running SPLASH-3 benchmarks, where T-bars represent the analytical value and solid bars are for the observed WCL among all the requests.

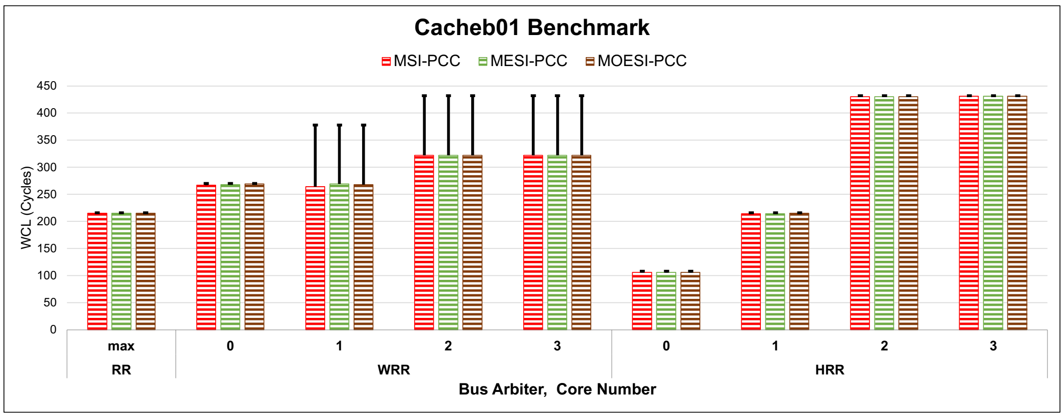

Figure 7 The observed WCL (solid bars) and the analytical WCL (T-bars) for Cacheb01 benchmark from EEMBC suite.

4-core system where each core has 16KB direct-mapped L1 cache, and the cores are connected to each other and to a last level cache by a snooping unified bus. The access latency of L1 (L_{Hit}) is 1 cycle, and the complete access slot to the bus is 54 cycles: 4 cycles for the request latency and 50 for the data transfer. To avoid the unnecessary large latency of the off-chip memory, we used a perfect last level cache which fits the whole data of the running application. All the experiments were carried out using an open-source cache simulator [1], which enables us to run trace-based simulations on memory traces collected using Intel's PINtool upon executing the benchmarks. We used SPLASH-3 benchmarks [26], which were configured to execute in 4 threads where each thread runs on a separate core. Moreover, we used benchmarks from the EEMBC [25] suite to simulate the extreme case of data sharing and coherence interference by running four instances of the same benchmark trace on the four cores, simultaneously. Accordingly, all the cores issue almost the same sequence of requests and share 100% of their data.

[1] `https://gitlab.com/FanosLab/pcc-sim`

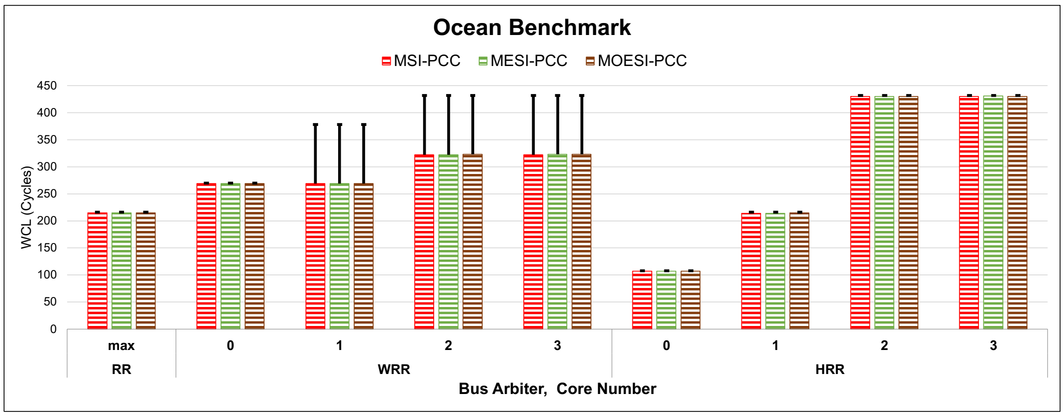

Figure 8 The observed WCL (solid bars) and the analytical WCL (T-bars) for Ocean benchmark from SPLASH-3 suite.

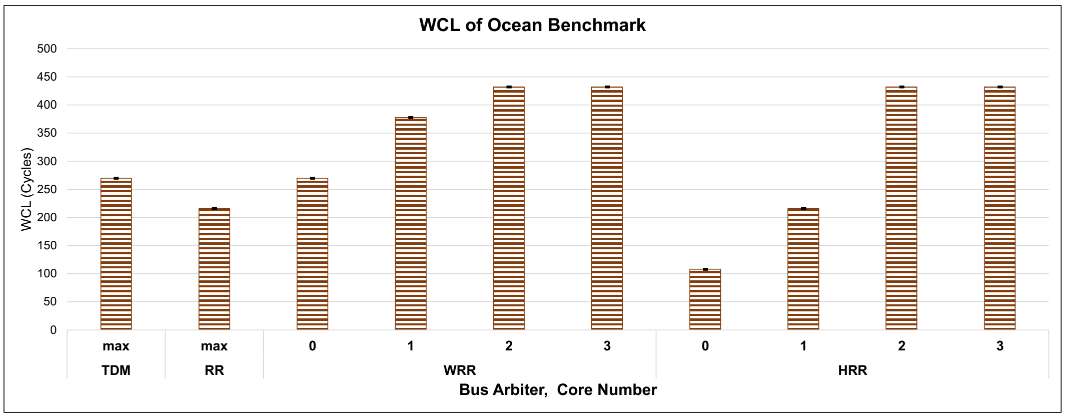

Figure 9 The observed WCL (solid bars) and the analytical WCL (T-bars) for Ocean benchmark from SPLASH-3 suite running in a 4-core system with OoO pipeline and $\mathcal{MOESI}$ coherence protocol.

7.1 Per-Request WCL

In this group of experiments, we measure the latency that is incurred by each memory request of the running benchmark, and compare it to the analytical WCL to ensure the safety and the tightness of the bounds. Figure 5 shows the results of running EEMBC benchmarks, and it compares between PMSI [15], PMESI [18], PMSI*, and PMESI*[19]. Besides, the figure includes three variants of PCC with the protocols $\mathcal{MSI}$, $\mathcal{MESI}$, and $\mathcal{MOESI}$ along with TDM bus arbiter. TDM is chosen for this experiment to have a fair comparison with the other solutions which adopt a TDM arbiter. Similarly, Figure 6 shows WCL of running the SPLASH benchmarks.

▶ **Observation 1.** *The observed WCLs of the solutions that implement cache-to-cache data transfer are much tighter compared to PMSI and PMESI. Further, the analytical bounds of PMSI and PMESI are quite pessimistic, which is clear from Figure 6 where hardly 50% of the bound is reached. Contrarily, the other solutions, including* PCC*, show notably tight bounds with both benchmarks suites.*

Figures 7 and 8 delineate WCL of chosen benchmarks from EEMBC and SPLASH, respectively. The results of the other benchmarks are consistent with Figures 7 and 8; hence, we include the results of a single benchmark from each suite for conciseness. In this experiment, a

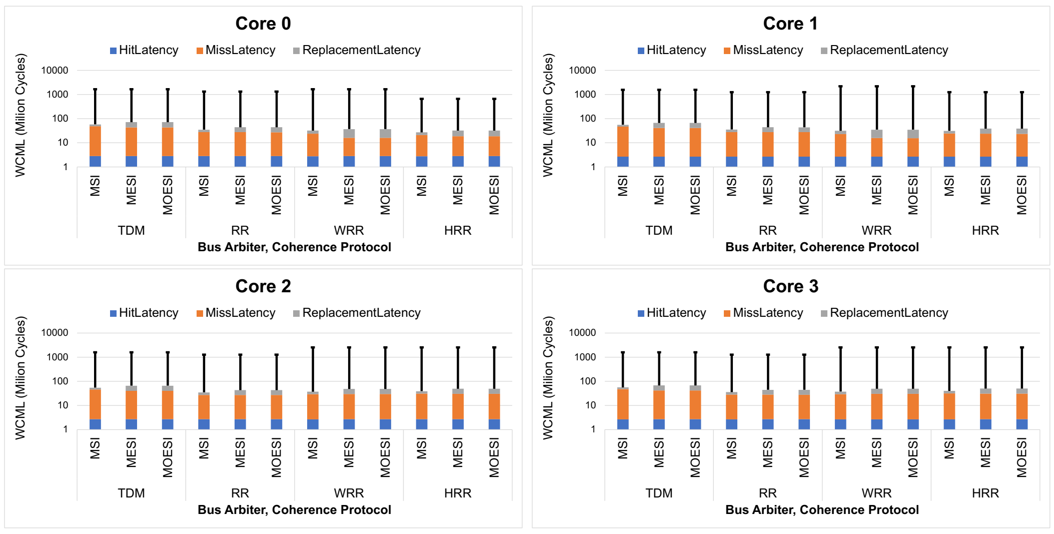

Figure 10 The total memory latency of running Ocean benchmark from SPLASH-3 suite. The T-bars represent the analytical WCML calculated using Equation 3 and the solid bars are for the observed total memory latency. The vertical axis is in a logarithmic scale.

comparison is held between a combination of configurations that contain $\mathcal{MSI}$, $\mathcal{MESI}$, and $\mathcal{MOESI}$ protocols along with RR, WRR, and HRR arbiters. WCLs of WRR and HRR are reported per core due to their unfair arbitration scheme; therefore, each core has a different bound based on its weight. The weights of the cores, in this and the following experiments, are {4, 2, 1, 1}, and the numbers are chosen arbitrarily to show the effect of the different weights. On the other hand, RR arbiter treats all the cores similarly; thus, the maximum value of WCL among the cores is reported.

▶ **Observation 2.** *There is no noticeable difference between results of Figures 7 and 8, although EEMBC simulates a synthetic effect of all the data is shared. This indicates the tightness and safety of WCL bounds in the case of a synthetic benchmark, like in Figure 7, and in a normal case, like Figure 8. Regarding the arbiters, RR shows a better WCL compared to WRR, whereas its WCL is average compared to the smallest and largest WCLs of HRR. The effect of the weights of the cores in WRR is not highly reflected into the WCLs, and the analytical WCL appears to be loose. On the other side, WCLs of the cores in HRR are rather tight.*

To complete the study of WCL, the analytical bounds are tested in the case of OoO execution. Figure 9 shows WCL of running a chosen benchmark from SPLASH-3 suite (the same benchmark used in Figure 8.) The execution of the benchmark is held using cores with OoO pipeline with a maximum of 8 outstanding requests. Additionally, TDM, RR, WRR, and HRR are used along with $\mathcal{MOESI}$ coherence protocol for the comparison. $\mathcal{MOESI}$ is chosen in this set of experiments since it is considered the most advance protocol among $\mathcal{MSI}$ and $\mathcal{MESI}$.

▶ **Observation 3.** *The analytical bounds are still respected in the OoO execution; however, WRR shows tighter WCLs compared to the in-order experiment shown in Figure 8. The tightness of the WCLs is due to the ability of the OoO cores to issue multiple requests and benefit from their high weights; therefore, the cores with lower weights are pushed to their bounds.*

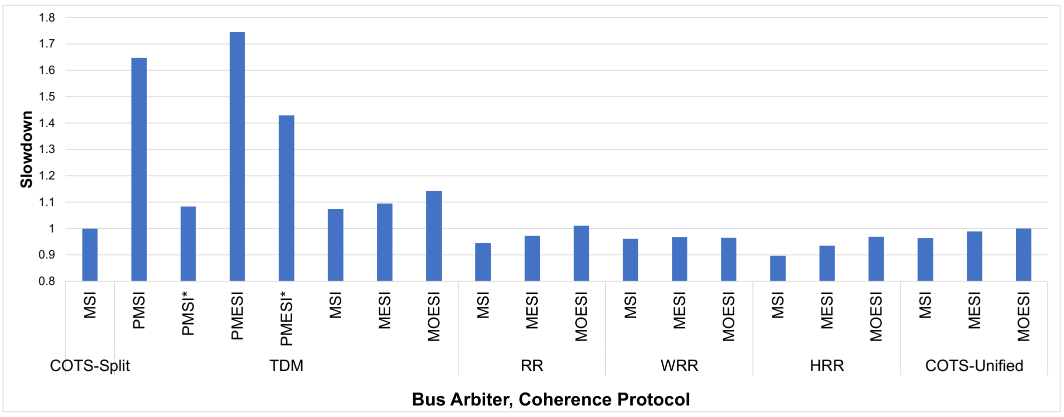

Figure 11 The average slowdown of the performance of the predictable cache solutions compared to COTS-Split and COTS-Unified. All the values of running EEMBC benchmarks are normalized to COTS-Split performance.

7.2 Total Task's WCL

In this experiment, we are concerned with the total memory latency that is incurred by each task. Similar to the previous experiment, a single benchmark from SPLASH-3 is run using $\mathcal{MSI}$, $\mathcal{MESI}$, and $\mathcal{MOESI}$ protocols along with TDM, RR, WRR, and HRR bus arbiters. Figure 10 shows WCML and the memory latency per each core. The latency is broken down into the three contributors to the latency: hit, miss, and replacement latencies. No information is assumed about the private vs shared data; and hence, WCML is calculated using the most conservative equation (Equation 3). Results are reported per core as each thread of the benchmark is mapped to a single core.

▶ **Observation 4.** *The gap between WCML and the observed latency is very large because of the pessimism in Equation 3. Regarding the protocols, $\mathcal{MSI}$ shows better latencies compared to $\mathcal{MESI}$ and $\mathcal{MOESI}$, and by looking at the replacement latency part, it is clear that replacements are the reason for the performance degradation of $\mathcal{MESI}$ and $\mathcal{MOESI}$. The justification of the higher number of replacements in $\mathcal{MESI}$ and $\mathcal{MOESI}$ can be returned to the addition of **E** and **O** states as they require a PutM message, unlike S state, before the eviction of a cache line. An optimization that can mitigate the latency of replacements is the empty PutM message (a PutM message that is not followed by a write-back) in the case of **E** state; however, this optimization has to be coupled with a dynamic arbiter, such as RR, to show an impact on the latency.*

7.3 Average-case Performance

In this section, we compare the average-case performance between different solutions including the solutions that implement COTS arbiters which favor performance only. Figure 11 delineates the average slowdown of each solution relative to a system that deploys an $\mathcal{MSI}$ protocol along with a split bus architecture and FCFS arbiter which is denoted by COTS-Split. This experiment runs the synthetic benchmarks of EEMBC to compare the performance of all the previous configurations and adding to them COTS-Unified configurations. COTS-Unified denotes a FCFS bus arbiter on top of a unified bus architecture. Similarly, Figure 12 shows the performance results of running SPLASH-3 benchmarks.

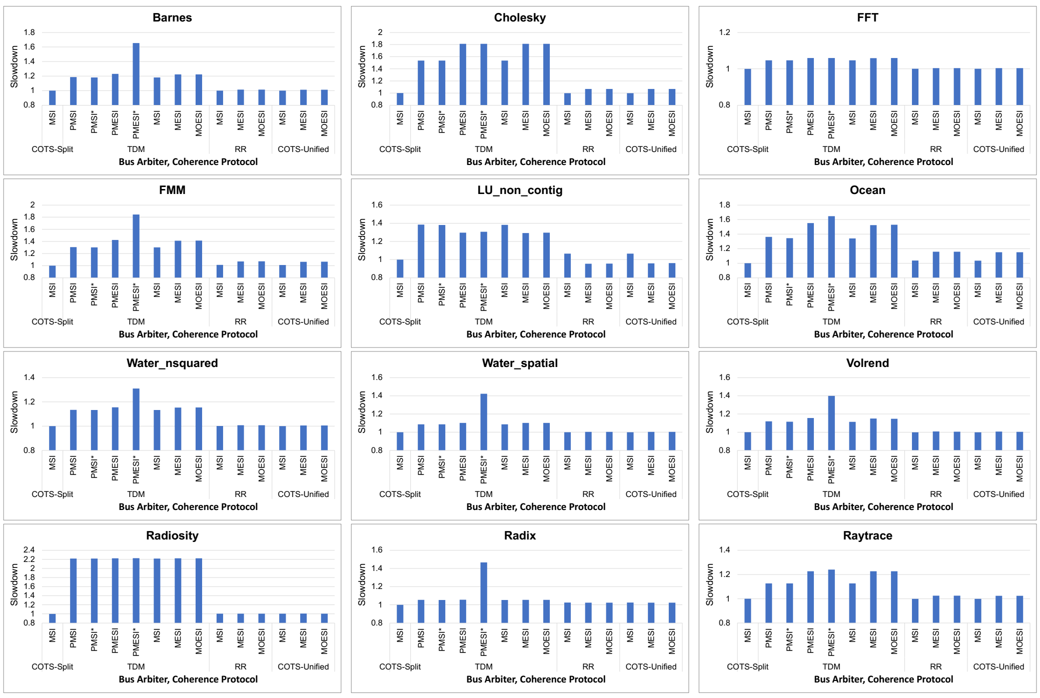

Figure 12 The slowdown of the performance of the predictable cache solutions compared to COTS-Split and COTS-Unified. All the values of running SPLASH-3 benchmarks are normalized to COTS-Split performance.

▶ **Observation 5.** *The relative performance of $\mathcal{MSI}$ protocol compared to $\mathcal{MESI}$ and $\mathcal{MOESI}$, shown in Figures 11 and 12, is consistent with their total memory latency which is discussed Observation 4. PMSI in Figure 11 shows a large performance degradation compared to $\mathcal{MSI}$ with TDM arbiter, while in Figure 12 the gap between them is minimal. The reason for this is the nature of the benchmarks as the percentage of the shared data in SPLASH is smaller than that in EEMBC; EEMBC has 100% of data shared to impose high coherence interference between cores. Also, we can make the same observation for PMESI and $\mathcal{MESI}$ with TDM. Moreover, PMESI* shows the worst performance in most of the benchmarks in Figure 12, and this is because PMESI* limits the cases that different cores can have the same cache line in **S** state. Finally, PCC deploying $\mathcal{MSI}$ with RR arbiter shows the best performance in Figure 12 and it is the closest to COTS-Split performance.*

The last experiment is to compare the performance of different arbiters that are coupled with $\mathcal{MOESI}$ protocol and OoO cores. Figure 13 shows the results of executing SPLASH-3 benchmarks, where the performance values of the arbiters are normalized to the values of COTS-Unified. The other predictable solutions are excluded from this experiments as none of them supports OoO execution.

▶ **Observation 6.** *The average gain in performance of $\mathcal{MOESI}$ with RR OoO execution compared to in-order execution is 5%. TDM shows the worst performance among the other arbiters, while WRR shows a slightly better performance than RR and HRR.*

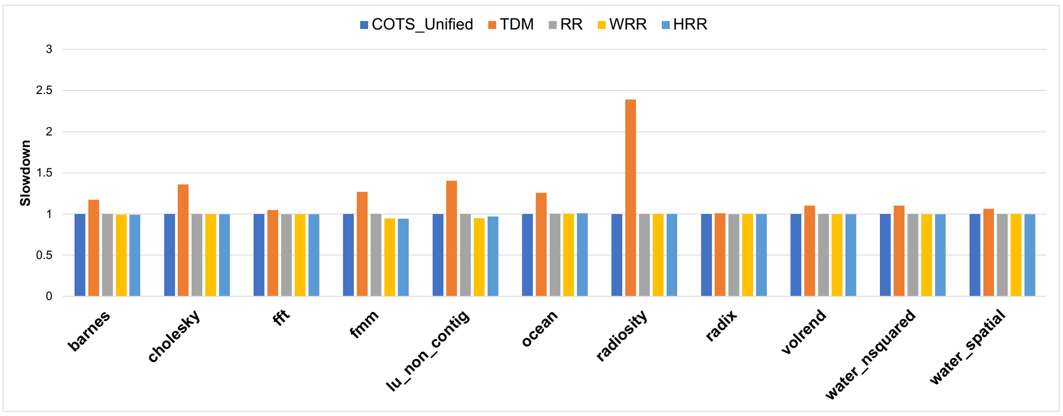

Figure 13 The slowdown of the performance of MOESI-PCC compared to COTS-Unified. All the values of running SPLASH-3 benchmarks are normalized to COTS-Unified performance.

8 Conclusion

We propose PCC: a solution to integrate COTS coherence protocols into legacy predictable real-time arbitration schemes without requiring any modifications to either of them. Doing so has several benefits. 1) PCC achieves tight latency bounds with minimal performance degradation. 2) It does not impose any burden on designing or verifying new protocols, which facilities adoption by industry. 3) It uses legacy arbitration schemes that have been studied for a long time, which in turn carries forward the credit of their analyzability making the proposed solution more appealing from a certification perspective. 4) It enables the integration of any coherence protocol with any predictable arbiter in a plug-and-play fashion. In this paper, we leveraged this capability to implement 3 different detailed coherence protocols as well as 4 commonly-used real-time arbiters. This allowed us to carry exploratory experiments for 12 different architectural configurations. Finally, we release the source code of the cycle-accurate implementation of such architectures for the community to use and expand.

References

1 ARM. ARM CoreLink CCI-550 Cache Coherent Interconnect, Technical Reference Manual, 2015. URL: `https://static.docs.arm.com/100282/0001/corelink_cci550_cache_coherent_interconnect_technical_reference_manual_100282_0001_01_en.pdf`.

2 ARM. Arm Cortex-A53 MPCore Processor Technical Reference Manual r0p4, 2016. URL: `https://developer.arm.com/documentation/ddi0500/j/Level-2-Memory-System/Snoop-Control-Unit`.

3 ARM. Arm Cortex-A9 MPCore Technical Reference Manual r4p1, 2016. URL: `https://developer.arm.com/documentation/100486/0401/snoop-control-unit`.

4 ARM. ARM Cortex-R82, Technical Reference Manual, 2021. URL: `https://developer.arm.com/documentation/101548/0002/?lang=en`.

5 Ayoosh Bansal, Jayati Singh, Yifan Hao, Jen-Yang Wen, Renato Mancuso, and Marco Caccamo. Cache where you want! reconciling predictability and coherent caching. *arXiv preprint*, 2019. `arXiv:1909.05349`.

6 Daniel Casini, Alessandro Biondi, Giorgiomaria Cicero, and Giorgio Buttazzo. Latency analysis of i/o virtualization techniques in hypervisor-based real-time systems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 306–319. IEEE, 2021.

7 Chun Chang. A Deep Dive on the QorIQ T2080 Processor, NXP, 2014.

8 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)*, 2016.

9 David Kruckemyer Craig Forrest. Arteris Ncore™ Cache Coherent Interconnect, Technology Overview, 2006.

10 Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A comparative study of predictable dram controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):1–23, 2018.

11 Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication centric design in complex automotive embedded systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

12 Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *2009 30th IEEE Real-Time Systems Symposium*, pages 68–77. IEEE, 2009.

13 Mohamed Hassan. On the off-chip memory latency of real-time systems: Is ddr dram really the best option? In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 495–505. IEEE, 2018.

14 Mohamed Hassan. Discriminative coherence: Balancing performance and latency bounds in data-sharing multi-core real-time systems. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

15 Mohamed Hassan, Anirudh M Kaushik, and Hiren Patel. Predictable cache coherence for multi-core real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–246. IEEE, 2017.

16 Mohamed Hassan and Rodolfo Pellizzoni. Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2323–2336, 2018.

17 Salah Hessien and Mohamed Hassan. The best of all worlds: Improving predictability at the performance of conventional coherence with no protocol modifications. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 218–230. IEEE, 2020.

18 Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. Designing predictable cache coherence protocols for multi-core real-time systems. *IEEE Transactions on Computers*, 2020.

19 Anirudh Mohan Kaushik and Hiren Patel. A systematic approach to achieving tight worst-case latency and high-performance under predictable cache coherence. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 105–117. IEEE, 2021.

20 Anirudh Mohan Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren Patel. Carp: A data communication mechanism for multi-core mixed-criticality systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 419–432. IEEE, 2019.

21 Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared data caches conflicts reduction for wcet computation in multi-core architectures. In *18th International Conference on Real-Time and Network Systems*, page 2283, 2010.

22 MILO MK MARTIN, MARK D HILL, and DANIEL J SORIN. Why on-chip cache coherence is here to stay. *Communications of ACM*, 2012.

23 Vijay Nagarajan, Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 15(1):1–294, 2020.

24 Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, 1995.

25 J. A. Poovey, T. M. Conte, M. Levy, and S. Gal-On. A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro*, 29(5):18–29, September 2009. doi:10.1109/MM.2009.74.

26 Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111. IEEE, 2016.

27 Nivedita Sritharan, Anirudh Kaushik, Mohamed Hassan, and Hiren Patel. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 433–445. IEEE, 2019.

28 Mohamed Younis and Mohamed Aboutabl. Communication handling in integrated modular avionics, October 3 2002. US Patent App. 09/821,601.