PISCOT: A Pipelined Split-Transaction COTS-Coherent Bus for Multi-Core Real-Time Systems

SALAH HESSIEN, McMaster University, Canada
MOHAMED HASSAN, McMaster University, Canada

Tasks in modern embedded systems such as automotive and avionics communicate among each other using shared data towards achieving the desired functionality of the whole system. In commodity platforms, cores communicate data through the shared memory hierarchy and correctness is maintained by a cache coherence protocol. Recent works investigated the deployment of coherence protocols in real-time systems and showed significant performance improvements. Nonetheless, we find these works to require modifications to commodity coherence protocols, assume simple in-order pipelines, and most importantly suffer from significant latency delays due to coherence interference along with average performance degradation. In this work, we propose PISCOT: a predictable and coherent bus architecture that (i) provides a considerably tighter bound compared to the state-of-the-art predictable coherent solutions (4× tighter bounds in a quad-core system). (ii) It does so with a negligible performance loss compared to conventional high-performance architecture coherence delays (less than 4% for SPLASH-3 benchmarks). This improves average performance by up to 5× (2.8× on average) compared to its predictable coherence counterpart. Finally, (iii) it achieves that without requiring any modifications to conventional coherence protocols. We show this by integrating PISCOT on top of two protocols with a detailed implementation with complete transient states: MSI and MESI.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: datasets, neural networks, gaze detection, text tagging

1 INTRODUCTION

Multi-core platforms are the norm nowadays in all computing systems, and real-time embedded systems are no exception. Multi-core platforms are envisioned to be the solution for the increasing computational and data demands in modern real-time embedded systems such as those deployed in automotive, avionics, and Internet-of-things (IoT). Nonetheless, multi-core platforms bring their own challenges. One of the biggest challenges is the interference among various cores in the system while competing to access shared hardware resources such as memory buses, shared caches, and off-chip memories. This interference hinders the system analyzability since the execution time of a task on one core now depends on the run-time behavior of tasks running on other cores. In order to provide the timing guarantees mandated by the real-time tasks, the hardware itself must be predictable such that the delays resulting from the aforementioned interference can be analytically bounded. To address this challenge, several efforts have been proposed to provide predictable memory buses [15, 17], shared caches [7, 11, 27, 35], and off-chip memories [9, 12, 16].

Despite being effective in managing the timing interference, most of these solutions assume that tasks are completely isolated with no communication among each other. We find this assumption to limit the applicability of these solutions in practical embedded systems, which require inter-task communication such as those deployed

Authors' addresses: Salah Hessien, salahga@mcmaster.ca, McMaster University, Hamilton, Ontario, Canada; Mohamed Hassan, mohamed.hassan@mcmaster.ca, McMaster University, Hamilton, Ontario, Canada.

1539-9087/2022/8-ART $15.00
https://doi.org/10.1145/3556975

in automotive [10], and avionics [43]. Consequently, recent approaches investigated the communication among tasks through shared data [2–4, 8, 13, 14, 20–22, 25, 36, 37, 40, 41]. Among these approaches, this paper is focusing on enabling tasks to communicate and share data by deploying hardware cache coherence, which is the approach followed by [2, 8, 13, 14, 20–22, 36, 37, 40, 41]. This is because cache coherence is the most commonly followed approach by commodity multi-core platforms [28], it improves overall system performance, and it does not impose any restrictions on the embedded legacy software or the operating system. In spite of their performance improvements over other real-time oriented data sharing techniques [41], current predictable coherence works suffer from several limitations. 1) They require major modifications to commodity coherence protocols (and hence, the hardware cache controllers). Those modifications are difficult to adopt by industry because of the significant time and intellectual effort required to implement and verify coherence protocols [32, 39]. 2) They are only considering in-order cores, where each core can have a maximum of one pending request at a time. 3) They suffer from extremely pessimistic worst-case latencies (WCLs) that reach to thousands of cycles for a single memory request to the shared cache as we explain in detail in Section 4. We studied in the prliminary version of this work [19] the deployment of coherence protocols using traditional predictable bus arbiters and investigate the sources of significant latency increase due to coherence interference (Section 4). Our study shows that most of the traditional arbitration schemes widely used in the real-time embedded systems domain are data-sharing oblivious. Therefore, to enable coherent data sharing while minimizing the coherence delays, we need a novel bus architecture that accommodates for this sharing by design. Motivated by this observation, we propose PISCOT, a predictable and coherent bus architecture. PISCOT resembles the following contributions. 1) It substantially reduces coherence delays, while improving overall system performance (Section 5). This is achieved by decoupling the data responses from their coherence requests, implementing a split-transaction interconnect with two separate buses. While the coherence requests are arbitrated using Time Division Multiplexing (TDM) to ensure predictability, the data responses are managed in a First Come First Serve (FCFS) fashion to increase average performance. Balancing the trade-off between predictability and performance is one of the main requirements of modern embedded systems. 2) Unlike existing solutions, PISCOT does not require any modifications to the underlying coherence protocol. This is key since modifications to coherence protocols are both hard to be adopted by commercial chips and hard to verify [2, 32, 39]. 3) To guarantee system predictability, we conduct a detailed timing analysis that formally provides an analytical upper bound for the latency suffered by any memory request. The derived bounds are 4× tighter than the state-of-the-art predictable coherent buses [14, 22, 41] for a quad-core system. 4) We deploy PISCOT in two different cache architectures currently adopted by commercial embedded systems. In the first, cores communicate only through the shared cache, while in the second, there is a direct cache-to-cache communication bus to increase efficiency.

This paper extends the preliminary work in [19] by making the following additional contributions. 1) One of the major aspects of PISCOT is that it works in tandem with conventional coherence protocols without requiring any modifications to them nor to the architecture of PISCOT. To show this aspect, we integrate PISCOT on top of two protocols with a detailed implementation including their complete transient state machines: the Modified-Shared-Invalid (MSI) and the Modified-Exclusive-Shared-Invalid (MESI) protocols. MESI is the protocol deployed in several embedded commercial-of-the-shelf (COTS) multi-core platforms such as NXP's T4080 with the quad-core E6500 architecture [6] that is commonly used in the avionics domain [33, 34], and ARM's advanced micro controller bus architecture (AMBA) with the coherent hub interface (CHI). Similar to the MSI integration, MESI's integration also includes different bus architectures: direct cache-to-cache and no cache-to-cache buses. 2) Since also PISCOT further enables simplifications in the coherence states due to its predicable nature, we added a section and a table discussing in detail how PISCOT can enable such simplifications (Section 5.2). 3) We also conduct extensive experiments to compare MSI and MESI protocol's behavior using both PISCOT and a commodity non real-time bus architecture as a baseline for comparison. Results confirm that PISCOT's latency bounds are independent of the underlying coherence protocol. 4) Another important aspect of PISCOT compared to existing

predictable coherence solutions is supporting both in-order and out-of-order cores (more details are in Section 5). In this paper, we study the effect of varying the possible number of pending requests (usually are the number of available entries in miss status handling registers (MSHRs)) on the system behavior (Section 7.5). 5) Finally, we conduct extensive sensitivity experiments to study the effect of the bus slot width (shared cache access time) on the predictability and performance of the system (Section 7.4).

We evaluate PISCOT with both the representative SPLASH-3 benchmarks as well as synthetic benchmarks. Comparisons with existing solutions show that PISCOT achieves up to 5× better performance (2.8× on average), while increasing memory bandwidth utilization by 12× on average across the SPLASH-3 benchmarks. We release the full implementation details as open-source for researchers to use and build on[1].

2 CACHE COHERENCE: A BACKGROUND

One of the key contributions behind PISCOT is that it offers predictable and coherent data-sharing without the need to apply any changes to the coherence protocol itself. Therefore, PISCOT operates in tandem with any underlying coherence protocol. In this paper, we exemplify by integrating PISCOT with both MSI and MESI protocols. MSI is the foundation of coherence protocols deployed in most existing architectures such as the MESIF protocol in Intel's i7 and the MOESI protocol in AMD's Opteron [18]. MESI also is commonly used in COTS mult-core platforms targeting embedded domains such as avionics and automotive. Examples include the NXP's T4080 [6] multi-core system-on-chip (SoC), and the ARM's CHI architecture.

Coherence State Machine. Table 1 lists the complete state machine with transitions of both MSI and MESI protocols. For MSI, there are three stable states for any cache line in a core's private cache; *modified* (`M`): meaning that the cache line is valid and modified (i.e. written), *shared* (`S`): meaning that it is valid but only read, and *invalid* (`I`): indicating a cache line that either does not exist in the private cache or has a stale data (a cache miss). A cache line can be in the `S` state in multiple cores' private caches. On the other hand, to maintain data correctness, only one core can have a cache line in the `M` state at any time, while all other cores will have this line in the `I` state. If a core has a load (store) miss to a cache line, it will issue a GetS() (GetM()) coherence message on the bus, and once it receives the data in its private cache, it moves to the `S` (`M`) state. A load to a cache line in the `S` or `M` state will be a cache hit. A store to a cache line in the `M` state is also a cache hit. Contrarily, a store to a cache line in the `S` state has to broadcast a coherence message on the bus (either a GetM() or an Upg() based on the deployed coherence protocol details) to inform other cores that might be in `S` state to invalidate their lines.

A core with a cache line in the `S` or `M` state that observes in the bus a GetM() message from another core to the same line (called OtherGetM()) has to move to the `I` state. If the core was in `M` state it also has to send the updated data to the shared memory and/or the requesting core based on whether there is a communication interconnect between private caches as we discuss later in this section. A core with the cache line in the `M` state upon observing a GetS() of another core (OtherGetS()) has to send the updated data similar to the previous situation, while moving to the `S` state.

Transient States. The aforementioned transitions between states do not usually happen atomically. They are usually interrupted by other requests from other cores as requests to the memory bus from different cores are allowed to interleave (i.e. there can be multiple pending requests at the same time) to increase system performance. For instance, a request can be pending for data to be fetched from the main memory; hence, the system allows for other younger requests to proceed if their data is already ready to increase overall throughput. During these interruptions of a request, the cache line may need to change its state to keep track of the updated coherence events on the bus, and this is the rule of *transient states*. Generally, a cache line moves to one or multiple transient state(s) in its journey from one stable state to another. In the interest of this paper, we classify transient states into four distinct categories.

[1] https://gitlab.com/FanosLab/piscot

Table 1. MSI and MESI Coherence Protocols with transient states. Grayed cells are the ones added by MESI, while red bold cells are the ones possible to be removed under PISCOT as Section 5.2 details. "X": an invalid (not possible) situation. "–": no action needs to be taken.

State	Core Event			Bus Event							
	Load	Store	Replace	OwnGetS	OwnGetM	OwnPutM	OtherGetS	OtherGetM	OtherPutM	Own data response	Own Excl
I	issue GetS/ IS^{ad}	issue GetM/ IM^{ad}	X	X	X	X	-	-	-	X	X
IS^{ad}	stall	stall	stall	-/IS^{d}	X	X	-	-	-	**–/IS^{a}**	X
IS^{d}	stall	stall	stall	X	X	X	-	-/ $IS^{d}I$	-	Hit/S	-/ IE^{d}
IE^{d}	stall	stall	stall	X	X	X	$IE^{d}S$	-/ $IE^{d}I$	-	Hit/E	X
$IE^{d}S$	stall	stall	stall	X	X	X	-	-/ $IE^{d}SI$	-	Hit SendData /S	X
$IE^{d}SI$	**stall**	**stall**	**stall**	**X**	**X**	**X**	-	-	-	**Hit+SendData / I**	**X**
$IE^{d}I$	stall	stall	stall	X	X	X	-	-	-	Hit SendData /I	X
IS^{a}	**stall**	**stall**	**stall**	**Hit/S**	**X**	**X**	**–**	**–**	**X**	**X**	**X**
$IS^{d}I$	stall	stall	stall	X	X	X	-	-	-	Hit/I	X
IE^{a}	**stall**	**stall**	**stall**	**Hit/E**	**X**	**X**	**–**	**–**	**X**	**X**	**X**
IM^{ad}	stall	stall	stall	X	-/IM^{d}	X	-	-	-	**–/IM^{a}**	X
IM^{d}	stall	stall	stall	X	X	X	-/ $IM^{d}S$	-/ $IM^{d}I$	-	Hit/M	X
IM^{a}	**stall**	**stall**	**stall**	**X**	**Hit/M**	**X**	**–**	**–**	**–**	**X**	**X**
$IM^{d}I$	stall	stall	stall	X	X	X	-	-	-	Hit SendData /I	X
$IM^{d}S$	stall	stall	stall	X	X	X	-	-/ $IM^{d}SI$	-	Hit SendData /S	X
$IM^{d}SI$	**stall**	**stall**	**stall**	**X**	**X**	**X**	-	-	-	**Hit+SendData / I**	**X**
S	Hit	issue GetM/ SM^{ad}	-/I	X	X	X	-	-/I	-	X	X
SM^{ad}	Hit	stall	stall	X	-/SM^{d}	X	-	-/ IM^{ad}	-	**–/SM^{a}**	X
SM^{d}	Hit	stall	stall	X	X	X	-/ $SM^{d}S$	-/$SM^{d}I$	-	Hit/M	X
SM^{a}	**Hit**	**stall**	**stall**	**X**	**Hit/M**	**X**	**–**	**–/IM^{a}**	**X**	**X**	**X**
$SM^{d}I$	Hit	stall	stall	X	X	X	-	-	-	Hit SendData /I	X
$SM^{d}S$	Hit	stall	stall	X	X	X	-	-/ $SM^{d}SI$	-	Hit SendData /S	X
$SM^{d}SI$	**Hit**	**stall**	**stall**	**X**	**X**	**X**	**–**	**–**	**–**	**Hit+SendData / I**	**X**
E	Hit	Hit/ M	issue PutM/ EI^{a}	X	X	X	Send Data/ S	Send Data / I	-	X	X
M	Hit	Hit	issue PutM/ MI^{a}	X	X	X	Send Data/ S	Send Data / I	-	X	X
MI^{a}	Hit	Hit	stall	X	X	Send Data/ I	Send Data/ II^{a}	Send Data / II^{a}	-	X	X
EI^{a}	Hit	stall	stall	X	X	Send Data/ I	Send Data/ II^{a}	Send Data / II^{a}	-	X	X
II^{a}	stall	stall	stall	X	X	-/I	-	-	-	X	X

- **Waiting for data and message states:** Those are marked with the superscript ^{ad} in Table 1. A core in these transient states has a pending request that is not granted access to the bus yet by the arbiter. Once the request message is issued on the bus, due to the reorderings and delays that can happen in the bus and its non-atomic nature, the core can first see either its coherence message or the requested data.
- **Waiting for data states:** Those are marked with the superscript ^{d} in Table 1. These states indicate that the core has already observed its coherence message but is yet waiting for data.
- **Waiting for message states:** Those are marked with the superscript ^{a} in Table 1. A core will be in one of these states if it receives its data before observing its coherence message.
- **Response to other requests states:** While a core is in one of the aforementioned three categories, it can observe requests from other cores (OtherGetM() or OtherGetS()) to the same cache line. Hence, it may need to move to another transient state to acknowledge the receiving of this request.

To illustrate the four categories, assume a load request to a cache line that is in the `I` state. Once the request misses in its private cache, the core queues a GetS() message to its local buffer waiting to be granted access to the bus by the arbiter. In this case (as Table 1 shows), the line will move to the $\texttt{IS}^{ad}$ state, which indicates that the core is waiting both to observe its message and receive the data in order to move from the stable `I` state to the stable `S` state. Afterwards, if the core observes its coherence message on the bus, it will change its state to $\texttt{IS}^{d}$ indicating that it is now waiting for data. On the other hand, if it receives the requested data before observing its coherence message, it has to change its state to $\texttt{IS}^{a}$ and wait for its message to appear on the bus. This is necessary since these broadcasted messages are the contract between all cores guaranteeing that they all observe changes to cache lines in the same order; otherwise, data inconsistencies will exist among cores. An example state from the fourth category happens if the core while in the $\texttt{IS}^{d}$ state, observes an OtherGetM() to the same cache line. As a result, it

has to move to the $\texttt{IS}^d\texttt{I}$ state. This state indicates that the core after receiving its requested data and conducting its load operation, has to invalidate its cache line since there is another pending store request from another core to the same cache line.

MESI: Exclusive State. MESI protocol optimizes over MSI by adding the *Exclusive* (E) state. A cache line in the E state is valid, non-modified, and exclusive (i.e. not cached by another core). The E state offers the following two main advantages. 1) A core that has a store request for a cache line in the E state hits silently without the need to access the shared bus or communicate with other cores. This is because E means that no other core has this line in its private cache. 2) In architectures where direct data transfer between cores' private caches is enabled, a core with a cache line in the E state can send this data directly to the requesting core. On the other hand, since stores to lines in the E state occur silently, a core that needs to evict a cache line in the E state can no longer do this silently (like the S state). Instead, it has to issue a write-back request, similar to the M state. As a result, the performance of MESI protocol compared to MSI is application-dependent. We provide an in-depth discussion about this performance with thorough experiments in Section 7.6. Since one of the major contributions of PISCOT is that it works in tandem with any underlying protocol, in this paper, we integrate PISCOT with both the MSI and MESI protocols.

Cache-to-Cache Communication. Two architectural models are considered with regard to how data is transferred among cores' private caches. The first model covers architectures that do not employ a direct cache-to-cache interconnect. In this case, the owner core always has to send the data to the shared memory (such as the last-level cache (LLC)). Afterwards, the shared memory sends this data to the requesting core. The second model represents architectures that support direct cache-to-cache communication. In this model, the owner core sends the data directly to the requesting core. In addition, if the requesting core's message was a GetS() (meaning that it is a load request), the owner also has to send the data to the shared memory since the shared memory will be the owner in this case. In both models, if there is no owner core (i.e. no core has the requested line in the modified state), the shared memory is the owner and it is responsible for sending the data to the requesting core.

3 RELATED WORK

Towards adopting multi-core platforms in real-time embedded systems, several proposals are introduced to predictably manage shared hardware components among cores [7, 9, 11, 15, 17, 27, 35]. Among these, two lines of work are closely related to this paper, memory bus arbitration, and cache coherence.

Predictable Bus Arbitration. The memory bus in a multi-core platform is one of the main sources of interference, which was found to solely increase the Worst-Case Execution Time (WCET) by up to 44% in a quad-core system [30]. to address this challenge, researchers proposed predictable bus arbitration schemes. This includes: Time Division Multiplexing (TDM) [17, 23], Round Robin (RR) [29], Harmonic RR (HRR) [42], and weighted [15] arbitration schemes. Unlike all these works, which focus only on timing interference assuming that tasks do not share data, PISCOT is a coherent bus that takes into account the cache coherence traffic and proposes a split-transaction architecture, where coherence requests and their responses are decoupled and arbitrated separately to increase system performance, while offering predictability.

Predictable Cache Coherence. There are multiple recent efforts to enable predictable sharing of data among real-time tasks through cache coherence [2, 8, 13, 14, 20–22, 33, 36, 37, 40, 41]. PISCOT differentiates itself from these works by enabling predictable cache coherence through its bus arbitration architecture without requiring any changes to the coherence protocol (such as in [14, 20–22, 40, 41]), the operating system's scheduler such as in [8], or the legacy software. The DISCO solution in [13] improves the WCL bounds by requiring a special handling of writes compared to reads. The works in [33, 36, 37] focus on modelling of cache coherence interference effects (formally in [36, 37] and experimentally in [33]) assuming an abstract model of existing commodity bus architectures. However, commercial architectures are not designed in the first place to be predictable, and thus,

Table 2. Arbiters

Approaches	Arbiter	Examples
COTS platforms baseline	High performance	FCFS [1, 24], split-transaction [5, 38, 45], priority-based [1, 31]
Traditional Real-Time Arbitration	Predictable by-design	TDM: [17, 23], RR: [29], Harmonic RR (HRR): [42], weighted RR: [15]
Data-Aware Arbitration	builds on traditional arbitration	PMSI [14], CARP [22], HourGlass [40], PENDULUM [41]
PISCOT	Predictable split-transaction	–

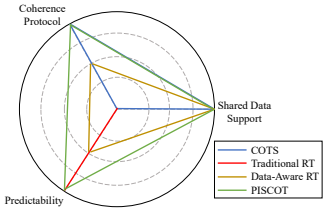


Fig. 1. Capabilities

provide only very pessimistic bounds if any. PISCOT, in contrast, is a predictable split-transaction coherent bus architecture that significantly reduces WCLs, while maintaining high average performance.

4 MOTIVATION

Three main observations motivate this work. 1) Existing solutions supporting coherence data sharing in commodity platforms are designed for performance. Accordingly, they provide no timing guarantees, and thus, cannot be safely used in real-time systems. 2) Traditional real-time arbiters designed for predictability are not considering data sharing among tasks, and it has been shown that even when using such predictable arbiters, they can lead to unpredictable behaviors when considering such sharing using coherence [14]. 3) Recent solutions that support coherence sharing of data are building on top of these traditional arbitration schemes. This leads to two significant drawbacks in these solutions. First, despite achieving predictability, the guaranteed latency bounds are notoriously large (in the range of thousands of cycles for a single request) [14, 22] which can be infeasible for systems with tight timing requirements. Second, they support coherence by proposing amendments to existing coherence protocols, which handicaps their adoption by industry in commercial platforms. We summarize these observations in Figure 1 and Table 2, and we discuss them in details in the following subsections.

4.1 Commodity Performance-Oriented Arbitration

Arbitration among different requests in COTS platforms is usually realized using a high-performance arbiter that favors system performance over other metrics such as fairness and predictability. Such arbiter prioritizes requests based on their arrival time (age-based priority), where older requests are serviced before younger ones. A common example of such arbiter is the First-Come First-Serve (FCFS) scheme [1, 24]. Such arbitration is not predictable since it provides no latency guarantees upon accessing the shared memory. This is because one core can have a request that is pending (theoretically) forever, while other cores are saturating the queues. In addition to age-based arbitration, some COTS platforms also deploy another level of fixed-priority arbitration to give higher-priority for requests from a certain processor. This also entails no guarantees are granted to lower-priority requests. A final observation about COTS arbiters is that for cache coherent systems, the bus is usually implemented as a split-transaction interconnect to increase system performance by concurrently handling both coherent requests (messages) and data responses [5, 38, 39]. For instance, the ARM Corelink CCI550 dictates separate channels for snooping requests and their corresponding responses. Similarly, the Intel's QPI designates different virtual channels to data and coherence messages.

4.2 Traditional Real-Time Arbitration

In multi-core real-time systems, access to the shared memory (e.g. the Last-Level Cache (LLC)) is managed through a predictable arbiter such as (TDM) [17, 23], and Round Robin (RR) [29]. Considering the TDM arbitration example depicted in Figure 2, a request suffers a maximum latency of one TDM period before it is granted access to the bus.

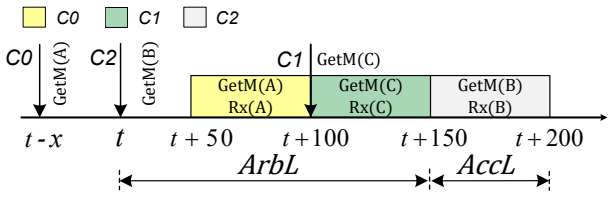


Fig. 2. Traditional TDM arbitration with no shared data.

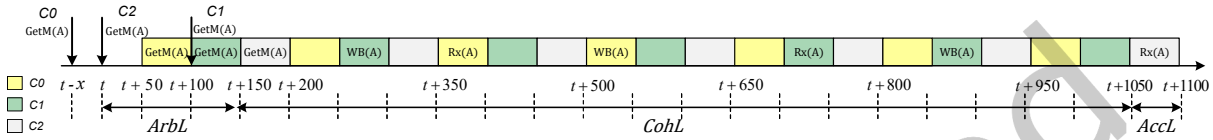


Fig. 3. TDM-based coherence approach [14]. Initially, C1 owns A in the M state.

For a system with N cores, this is $N \cdot S$ cycles, where S is the slot width in cycles. This occurs when the requesting core just misses its own slot. Please note that throughout this section, we denote a core as Cx, where x is the core index. The GetM(B) from $C2$ in Figure 2 is an example of such a request, where it arrives to the private cache controller at timestamp t. Assuming that $C2$ just missed its own slot, it waits until $t + 150$ to gain access to the bus. Since the system in Figure 2 has three cores, this is equivalent to a one TDM period of 3 slots assuming that the slot width allows for only one memory transfer (one request) and is 50 cycles. Once granted access to the bus, the request conducts its memory transfer consuming an extra slot (50 cycles) and finishes at $t + 200$.

The big limitation of this analysis is that it only applies if cores do not share data. In the example in Figure 2, all the cores request to access different cache lines. Consequently, the shared memory is able to respond with the correct data in the request's same slot. Unfortunately, this does not apply if cores are allowed to share data. It has been shown by [14] that shared data can lead to unpredictable behavior even when deploying a predictable arbitration such as TDM.

4.3 Coherent Shared-Data Aware Predictable Arbitration

To guarantee predictability while allowing coherent sharing of data, several recent arbitration solutions have been proposed [13, 14, 20–22, 41]. All these solutions assume a variant of the TDM arbitration scheme and propose coherence protocol as well as architectural changes to support predictability. Despite showing that coherence can lead to significant performance improvements in data-sharing real-time systems, they incur significant WCL bounds. To illustrate this drawback, Figure 3 delineates the TDM behavior for the same system in Figure 2 but with assuming that cores can share data, and hence, they issue requests to the same cache line, A. The example follows the protocol guidelines from PMSI [14]. It is clear from Figure 3 the significant added latency due to the coherence interference on the shared data. The request under analysis (GetM(A) from $C2$) in this case has to wait for every other core to receive the data of cache line A, conduct the store operation, and then write it back to the shared memory. Since the slot width of the TDM allows for only one memory transfer, and every core gets one slot per TDM period, every core now requires two TDM periods to conduct the aforementioned operation. As a result, $C2$'s GetM(A) request waits until timestamp $t + 1050$ in Figure 3 before it can start receiving its requested data. Formally, for a system with N cores and a TDM arbitration with shared data, a request has to wait for up to $(2 \cdot N^2 + 2 \cdot N) \cdot S$ before it can start transferring its requested data [14]. The other existing solutions while supporting systems with mixed criticalities [22, 41], this comes at the expense of incurring even larger WCL than PMSI if all cores have the same criticality.

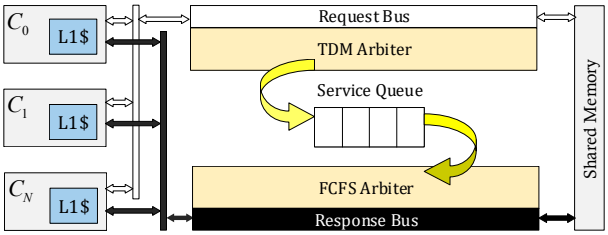


Fig. 4. PISCOT architecture example by applying it between private L1 caches and shared memory.

It is worth noting that in Figure 3 it might seem that there are many idle slots, and thus, this large latency can be completely avoided using a work-conserving schedule. However, this is not true since there can be requests from other cores in the system that utilize these slots. They are not shown in Figure 3 for simplicity. For example, $C0$ receives its requested data at timestamp $t + 400$. Thus, it can issue another request afterwards in its coming slots. Clearly, in an out-of-order architecture, more pending memory requests can also co-exist in the system.

Two key observations we make in this paper about the existing predictable cache-coherent TDM-based solutions. First, their previously highlighted large WCLs are mainly because they inherit the scheduling paradigm of traditional real-time arbiters (such as TDM in this case but the argument applies to other arbiters such as RR). This paradigm when applied to systems with shared data, it couples two different types of communication into the same bus arbitration. Namely, it couples both coherence messages and data transfers and schedules them using the same bus arbitration, which is inherited from traditional non-data-sharing TDM schedules. This in addition to the fact that the TDM slot has to accommodate for at least one memory transfer to be efficient to service ready memory requests, leading to the excessively large memory delays when introducing data sharing. Second, they impose certain modifications to the coherence protocol to enable predictability. As previously discussed, modifications to coherence protocols are highly costly in terms of verification and are thus inconceivable to adopt by industry.

Based on these observations, PISCOT targets to enable data sharing in real-time systems, while significantly reducing the associated coherence delays by decoupling the two different communication types. This is achieved by using a split-bus architecture, where requests (through coherence messages) and responses (i.e. data transfers) are issued in different buses and are managed using different arbitration mechanisms. In addition, PISCOT does not impose any changes to existing coherence protocols; therefore, disburden system designers from the need to re-verify the coherence protocol.

5 PROPOSED SOLUTION

In this section, we detail the architectural details of PISCOT, which Figure 4 delineates its high-level modules. Figure 4 shows PISCOT managing accesses between private L1 caches and shared memory. The details of the deployed coherence protocol is not shown since as stated before PISCOT operates independent of these details and hence any COTS coherence protocol can be assumed to be implemented in these caches. Compared to the solutions discussed in Section 4, PISCOT makes multiple architecture decisions to take into account predictability by design, while maintaining a high average-case performance. 1) PISCOT's architecture migrates from the traditional arbitration schemes considered by the community (such as TDM and RR) to a split-transaction bus interconnect that connects private caches and the shared memory as Property 1 explains.

PROPERTY 1. PISCOT *implements a split-transaction bus through deploying two buses: a* `Request Bus` *and a* `Response Bus`*. The* `Request Bus` *is responsible for broadcasting the coherence messages initiating memory requests, while the* `Response Bus` *transfers data as a response to these requests.*

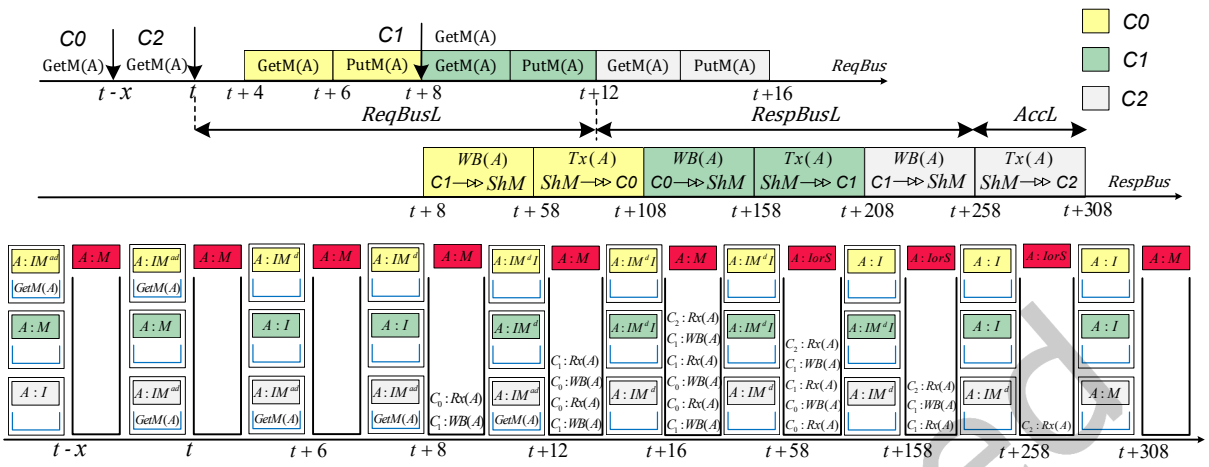


Fig. 5. An illustrative example for the operation of PISCOT. Latency components are for the getM(A) request from $C2$. At different time instances: the bottom of the figure shows the state of the private core's cache line (left side), shared memory state (marked in red), and the `Service Queue` contents on the right side.

2) Aiming at performance, the `Request Bus` and the `Response Bus` operate in parallel. On the other hand, to simplify system analysis and maintain predictability, both buses communicate through only one module: the `Service Queue`. Requests broadcasted on the `Request Bus` are buffered into the `Service Queue` until they are selected by the `Response Bus`'s arbiter. 3) Unlike conventional solutions that use high-performance arbiters at the expense of predictability (e.g. FCFS), the `Request Bus` in PISCOT is managed using a TDM arbiter to predictably manage interference among different cores (Property 2). To increase system performance, a work-conserving TDM is deployed, where at any slot, if the dedicated core does not have a ready request, the arbiter picks the next core with a pending request instead of leaving the slot idle as in traditional non work-conserving TDM. The TDM slot allows for one coherence (demand) request from the core that owns the slot, in addition to any response message to this demand request. Under the implemented snooping-based protocol, a maximum of only one response is possible as a result of a demand request. This is usually happens if there is an owner core that indicates it needs to send the data as a response to the original demand request (`PutM` message).

PROPERTY 2. PISCOT *manages the* `Request Bus` *using a work-conserving TDM arbiter.*

4) The `Response Bus`'s arbiter implements a First-Come First-Serve (FCFS) scheduler, and thus, serves requests based on their arrival time on the `Service Queue` (Property 3). The oldest request will be at the head of the queue, and therefore, is serviced first by the FCFS response arbiter. Once selected by the FCFS arbiter, the requested data is transferred on the `Response Bus` to the requesting core's private cache, the request message is removed from the `Service Queue`, then the core proceeds with its load/store operation indicating that the request is successfully finished.

PROPERTY 3. PISCOT *manages the* `Response Bus` *using a FCFS arbiter.*

5) An important contribution of PISCOT compared to existing predictable coherence solutions is that it supports out-of-order execution. PISCOT does so by allowing cores to have multiple outstanding requests, while distinguishing between two different types of outstanding requests from cores to the coherent cache hierarchy: pending requests (not issued yet, Definition 5.1) , and in-service requests (issued but not finished yet, Definition 5.2). While

PISCOT allows a core to have multiple of the former (usually decided by the number of MSHR entries), it limits the later to be one per core.

Definition 5.1. (**A Pending Request**) is a request that misses in the core's private cache and waiting to be issued by the arbiter to the `Request Bus` to be serviced.

Definition 5.2. (**An In-Service Request**) is a request that misses in the core's private cache and is granted access to the shared `Request Bus` but is still waiting for its data response.

According to Property 4, outstanding requests from a certain core would remain in its local buffer and will not be picked by the TDM arbiter if the core already has one request in-service (i.e. queued in the `Service Queue`). The rationale for this is to limit the coherence interference among cores such that a request from any core can suffer interference due to a maximum of only one request from each other core, which leads to tightening worst-case latencies and minimizing interference from other cores compared to the conventional MSI protocol with FCFS split-transaction bus as we detail in the latency analysis in Section 6.

PROPERTY 4. PISCOT *supports Out-of-Order (OOO) architectures by allowing cores to issue multiple outstanding requests. However, to limit coherence interference, it only services at most one request from any given core at a time.*

It is crucial to point out that although PISCOT limits in-service requests to one per core, it is still can leverage the OOO significant performance improvement over in-order cores supported by the previous predictable coherent protocols. This is because with this distinguish between pending and in-service requests, PISCOT allows private cache hits OOO and independent of the pending requests, which enables the usage of non-blocking caches and hits-under-misses.

5.1 Illustrative Example

To better explain the operation of PISCOT, we use the same example from Section 4 for a system with three cores: $C0$–$C2$ and delineates PISCOT's behavior in Figure 5. The example focuses on a single cache line A, which is assumed to be initially owned by $C1$. At timestamp $t - x$, a store request to A from $C0$ misses in its private cache (it was originally in `I` state). As a result, a GetM(A) message is placed in its cache controller's local buffer waiting for $C0$'s slot on the request bus. The line state changes in the private cache from `I` to $\texttt{IM}^{ad}$ waiting for its message to appear on the bus. The same situation occurs for $C2$ at timestamp t. At $t + 4$, $C0$ is granted a slot by the TDM arbiter and its request is issued on the `Request Bus`. The coherence message is assumed to consume two cycles to be broadcasted. Accordingly, $C0$ observes its OwnGetM(A) on the bus and move to $\texttt{IM}^{d}$ while waiting for data. On the other hand, once $C1$ observes $C0$'s GetM(A) (OtherGetM(A)) and since $C1$ is the owner of A, it responds with placing the updated data in its local buffer to be written back to the shared memory (timestamp $t + 6$) and moves to `I` state. In addition, two actions are pushed into the `Service Queue` as a result of $C0$'s request. This is because $C1$ has to write back its updated A first to the shared memory and then the shared memory will send the data to $C0$; these are indicated in Figure 5 in the `Service Queue` as $C1$:WB(A) and $C0$:Rx(A), respectively.

Simultaneously at $t + 8$, a GetM(A) request from $C1$ arrives and is issued on the `Request Bus` immediately since it is $C1$'s slot. Similar to what happened during $C0$'s slot, $C1$ moves to the $\texttt{IM}^{d}$ state and two actions are pushed into the `Service Queue`: $C0$:WB(A) and $C1$:Rx(A). The reason for this is that $C0$ should obtain its requested data first, according to the FCFS schedule, conduct its store operation, and write back the updated data to the shared memory before $C1$ can proceed with its GetM(A) request. For the same reason, $C0$ moves to the $\texttt{IM}^{d}\texttt{I}$ state. This is indicated at timestamp $t + 12$. Now, $C2$ is finally granted access to the `Request Bus` and issues its request. Similar events to those during $C1$'s slot occur with the difference that $C1$ is the owner responsible to write-back A before the shared memory sends it to $C2$ according to the FCFS order. For the `Response Bus`, it

services requests in the Service Queue in order of their arrival as previously explained. Assuming that one data transfer requires 50 cycles, it finishes the data transfer of $C1$'s WB(A) to shared memory at $t + 58$. $C0$'s Rx(A) from shared memory at $t + 108$, performs its store operation and places the new data in its local buffer and moves to I state. $C0$'s WB(A) to shared memory finishes at $t + 158$. $C1$'s Rx(A) from shared memory at $t + 208$, performs its store operation and places the new data in its local buffer and moves to I state. $C1$'s WB(A) to shared memory finishes at $t + 258$, and finally $C2$ receives A from shared memory at $t + 308$.

Comparing this with the behavior of PMSI adopting the traditional TDM bus in Figure 3, it shows the clear advantage of PISCOT that reduces the total latency of the same sequence of memory requests by 792 cycles (from $t + 1100$ to $t + 308$). More detailed comparisons on the effect of both WCL as well as average performance are introduced in Section 7.

5.2 Coherence Protocol Simplification

As we discussed earlier, a key aspect of PISCOT is that it enables predictably and coherently sharing data without any modifications to the underlying coherence protocol itself. Therefore, PISCOT arbitration mechanism explained so far operates in tandem with any coherence protocol. Moreover, our investigation shows that deploying PISCOT can in fact simplify the coherence protocol by removing some of its transient states due to its predictable architecture. In this section, we iterate through the states that are no longer needed upon adopting PISCOT as the bus arbitration mechanism, while the underlying coherence protocol is MSI or MESI. Before doing so, it is worth noting however that, this does not mean that PISCOT obligates the removal of those transient states since as aforestated, PISCOT does not require any changes to the coherence protocol. Instead, if one was to design a coherence protocol with PISCOT, those states become obsolete. To verify this theory, in our implementation of the detailed MSI and MESI protocols, we adopted two versions: 1) the original protocol with all the states including those that are not necessary (i.e., fully implementing Table 1), and 2) the protocol with only the necessary states. States and transitions that can be removed under PISCOT are indicated in red bold text in Table 1, while the new simplified protocol (under both MSI and MESI) is shown in Table 3. Confirming our analysis, simplified protocols exhibit the exact same behavior as their original counterparts and the obsolete states are not observed at all during both verification and execution. We now detail those unnecessary states.

Two types of transient states are not necessary under PISCOT. First type is the *waiting for message states* explained in Section 2. This includes three states for MSI: $\texttt{IM}^a$, $\texttt{IS}^a$, and $\texttt{SM}^a$, and a forth one in MESI: $\texttt{IE}^a$. Based on the discussed operation of PISCOT, this situation will never occur for the following reason. A data transfer occurs on the Response Bus only when its corresponding message is picked up from the Service Queue by the FCFS arbiter. However, a message is only queued into the Service Queue when it is issued on the Request Bus, which also means that the message is observed by the requesting core. Therefore, a core will never receive its requested data before observing its own message. This reflects in Table 3 as marking these scenarios as impossible ("X").

Second type includes the states that track the fact that there are multiple pending requests from other cores to the same cache line. This includes two states for MSI: $\texttt{SM}^d\texttt{SI}$, and $\texttt{IM}^d\texttt{SI}$ and a third one in case of MESI: $\texttt{IE}^d\texttt{SI}$. These states indicate that while pending for data to write, the core first observed an OtherGetS(). As a result, eventually after conducting the store (or exclusive read) operation, it has to send data and then move to S state, and then observed another OtherGetM() message, and hence, the core must eventually invalidate. Those states are originally needed for the core to keep track of the order of those other messages and to know once received its data and performed its operation, which core to send the data to. In PISCOT, nonetheless, keeping order of requests is simply maintained by the Service Queue. Accordingly, the core does not need to track the order of other messages, and it only needs to track what state to move to eventually after receiving its data. Therefore, as indicated by Table 3, if a core is in the $\texttt{SM}^d\texttt{S}$ state and observes an OtherGetM(), it can move to $\texttt{SM}^d\texttt{I}$ instead of $\texttt{SM}^d\texttt{SI}$. Similarly, if a core is in the $\texttt{IM}^d\texttt{S}$ state and observes an OtherGetM(), it can move to $\texttt{IM}^d\texttt{I}$ instead of $\texttt{IM}^d\texttt{SI}$. This removes the need to these two transient states in MSI. Similarly for MESI, if a core is in the $\texttt{IE}^d\texttt{S}$

Table 3. Simplified MSI and MESI protocols under PISCOT. Grayed cells are the ones added by MESI. "X": an invalid (not possible) situation. "−": no action needs to be taken.

State	Core Event			Bus Event							
	Load	Store	Replace	OwnGetS	OwnGetM	OwnPutM	OtherGetS	OtherGetM	OtherPutM	Own data response	Own Excl
I	*issue GetS/* IS^{ad}	*issue GetM/* IM^{ad}	X	X	X	X	-	-	-	X	X
IS^{ad}	*stall*	*stall*	*stall*	*-/*IS^{d}	X	X	-	-	-	X	X
IS^{d}	*stall*	*stall*	*stall*	X	X	X	-	*-/* $IS^{d}I$	-	*Hit/S*	*-/* IE^{d}
IE^{d}	*stall*	*stall*	*stall*	X	X	X	$IE^{d}S$	*-/* $IE^{d}I$	-	*Hit/E*	X
$IE^{d}S$	*stall*	*stall*	*stall*	X	X	X	-	*-/* $IE^{d}I$	-	*Hit SendData /S*	X
$IE^{d}I$	*stall*	*stall*	*stall*	X	X	X	-	-	-	*Hit SendData /I*	X
$IS^{d}I$	*stall*	*stall*	*stall*	X	X	X	-	-	-	*Hit/I*	X
IM^{ad}	*stall*	*stall*	*stall*	X	*-/*IM^{d}	X	-	-	-	X	X
IM^{d}	*stall*	*stall*	*stall*	X	X	X	*-/* $IM^{d}S$	*-/* $IM^{d}I$	-	*Hit/M*	X
$IM^{d}I$	*stall*	*stall*	*stall*	X	X	X	-	-	-	*Hit SendData /I*	X
$IM^{d}S$	*stall*	*stall*	*stall*	X	X	X	-	*-/* $IM^{d}I$	-	*Hit SendData /S*	X
S	*Hit*	*issue GetM/* SM^{ad}	*-/I*	X	X	X	-	*-/I*	-	X	X
SM^{ad}	*Hit*	*stall*	*stall*	X	*-/*SM^{d}	X	-	*-/* IM^{ad}	-	X	X
SM^{d}	*Hit*	*stall*	*stall*	X	X	X	*-/* $SM^{d}S$	*-/* $SM^{d}I$	-	*Hit/ M*	X
$SM^{d}I$	*Hit*	*stall*	*stall*	X	X	X	-	-	-	*Hit SendData /I*	X
$SM^{d}S$	*Hit*	*stall*	*stall*	X	X	X	-	*-/* $SM^{d}I$	-	*Hit SendData /S*	X
E	*Hit*	*Hit/ M*	*issue PutM/* EI^{a}	X	X	X	*Send Data/ S*	*Send Data / I*	-	X	X
M	*Hit*	*Hit*	*issue PutM/* MI^{a}	X	X	X	*Send Data/ S*	*Send Data / I*	-	X	X
MI^{a}	*Hit*	*Hit*	*stall*	X	X	*Send Data/ I*	*Send Data/* II^{a}	*Send Data /* II^{a}	-	X	X
EI^{a}	*Hit*	*stall*	*stall*	X	X	*Send Data/ I*	*Send Data/* II^{a}	*Send Data /* II^{a}	-	X	X
II^{a}	*stall*	*stall*	*stall*	X	X	*-/I*	-	-	-	X	X

state and observes an OtherGetM(), it can move to $\texttt{IE}^{d}\texttt{I}$ instead of $\texttt{IE}^{d}\texttt{SI}$. Note that this type of states can include more than two states in more complicated protocols, which further increases the simplification benefit of PISCOT for such protocols.

It is also important to mention that this simplification does not affect by any means the correctness of the coherence protocol for data sharing. This can be reasoned about from two angles. First, from theory perspective, by construction, the removed states cannot be visited at all under the operation of PISCOT as detailed in the previous explanation. Second, from empirical verification perspective, as aforementioned, we implement both the full state machine of the protocol as well as the simplified one and we run all our evaluation using both. For all experiments including real benchmarks from the SPLASH-3 suite as well as data-stressing synthetic micro-benchmarks, both implementations show exactly the same behavior. States to be removed are never visited in the full protocol, while all other states and transitions are confirmed to be visited in both implementations. This empirically proves that the behavior of the simplified protocol under PISCOT is exactly the same as the full protocol.

5.3 Satisfying Coherence Predictability Invariants

Coherence protocols can generally lead to unpredictable behaviors if not carefully managed. In addition, previous works have shown that combining conventional coherence protocols with traditional predictable arbiters also breaks system's predictability [14]. Since we claim that PISCOT indeed achieves predictability by utilizing conventional coherence while deploying the proposed split-transaction predictable arbiter, we believe it is necessary to elaborate more on how PISCOT achieves this predictability. Authors of [14] introduced 6 invariants that they stated that they must be satisfied to ensure predictability in the existence of coherence. We now show how PISCOT, unlike PMSI [14], is satisfying those invariants without the need to modify the coherence protocol. This discussion also illustrates the novel operation of PISCOT compared to traditional predictable arbiters such as TDM when tasks can share data. For inclusiveness, we state each invariant and then prove how PISCOT satisfies it. We prove each case by contradiction starting with a hypothesis that PISCOT breaks such invariant and then show that this contradicts PISCOT's operation explained at the beginning of this section.

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 5.3. PISCOT *satisfies Invariant* 1.

PROOF. The proof is trivial since allowing a core to send a request without being granted access by the arbiter contradicts with PISCOT's TDM arbiter at the `Request Bus`. □

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

LEMMA 5.4. PISCOT *satisfies Invariants* 2.

PROOF. Let Req_i and Req_j be two requests to the same cache line such that Req_i arrived to the shared memory first. Assume that the shared memory serviced Req_j before Req_i such that Invariant 2 is broken. (1)
Now considering PISCOT's operation, Req_i will arrive to the shared memory first only if it is broadcasted on the `Request Bus` first. Hence, Req_i arriving at the shared memory first indicates that it has been queued into the `Service Queue` ahead of Req_j. Now, according to the `Response Bus`'s FCFS, Req_i must be serviced before Req_j. (2)
(1) and (2) contradicts, which completes the proof. □

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

LEMMA 5.5. PISCOT *satisfies Invariant* 3.

PROOF. Let $Req_i(A)$ and $Req_j(B)$ be two requests to cache lines A and B respectively that are owned by Core Ck such that Ck observes $Req_i(A)$ first. To break Invariant 3, PISCOT has to service $Req_j(B)$ before $Req_i(A)$. (1)
Now, according to PISCOT's operation, a core responds to a request for a cache line that it owns by placing the data immediately in its local buffer. Additionally, a WB action is queued into the `Service Queue` along with its initiating coherence message of the request itself during the same `Request Bus`'s TDM slot. For instance, at time $t+8$ in Figure 5, $C0$'s GetM(A) message resulted in pushing two actions to the `Service Queue`: 1) $C1$ has to write back A (WB(A)) first and only afterwards 2) $C2$ can receive its requested data (RX(A)) from shared memory. Since Ck observes $Req_i(A)$ first, it mandates under PISCOT that $Req_i(A)$ was issued in the `Request Bus` before $Req_j(B)$. Additionally, since requests are queued in the `Service Queue` based on their appearance timestamp on the `Request Bus`, it mandates that $Req_i(A)$ and its corresponding WB(A) are queued in the `Service Queue` ahead of $Req_i(B)$ and its WB(B). Finally, according to the `Response Bus`'s FCFS policy, $Req_i(A)$ will get its data before $Req_i(B)$. (2)
(1) and (2) contradicts, which completes the proof. □

INVARIANT 4. *A write request from a core that is a hit to a non-modified line in its private cache has to wait for the arbiter to grant this core an access to the bus.*

LEMMA 5.6. PISCOT *satisfies Invariant* 4.

PROOF. Let $Req_i(A)$ be a write request from core Ck to line A that Ck has in the S state in its private cache. To break Invariant 4, PISCOT shall allow $Req_i(A)$ to hit in the private cache and execute the operation silently without waiting for any permission from the bus arbiter. (1)
According to PISCOT's coherence protocol inherited from conventional MSI/MESI (Table 1), a store to a cache line in S state has to issue a getM() coherence message and wait in the SM^{ad} state. Afterwards, this message is only issued on the bus once its core is granted access according to the Request Bus's TDM schedule. (2)
(1) and (2) contradicts, which completes the proof. □

INVARIANT 5. *A write request from a core that is a hit to a non-modified line, A, in its private cache has to wait until all waiting cores that previously requested A get an access to A.*

LEMMA 5.7. PISCOT *satisfies Invariant* 5.

PROOF. Let cache line A to be initially in the S state in core Cj's private cache. Let also $Req_i(A)$ be a read request from core Ci to cache line A that is broadcasted on the bus at time $t1$. Then, assume that Cj at time $t1 + \delta$ (where $\delta > 0$) has a store request $Req_j(A)$ to A. To break Invariant 5, assume that $Req_j(A)$ is serviced before $Req_i(A)$. (1)
However, from Lemma 5.6, it follows that $Req_j(A)$ has to wait for Cj's TDM slot on the `Request Bus` to broadcast a $GetM(A)$ message on the bus before it can proceed with its store operation. Assume that this happens at time $t2$. Since $Req_j(A)$ arrived at $t + \delta$, it follows that $t2 \geq t1 + \delta$. As a result and from Lemma 5.4, $Req_i(A)$ request is serviced before $Req_j(A)$ since $t2 > t1$. (2)
(1) and (2) contradicts, which completes the proof. □

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

LEMMA 5.8. PISCOT *satisfies Invariant* 6.

PROOF. Assume a system with N cores C0 to CN such that one core Ci, $0 \leq i \leq N$, has a request to service from the memory, say Req_i, while all the other $N - 1$ cores keep issuing requests to cache lines that are modified (owned) by Ci. To break Invariant 6, Ci keeps servicing these requests and is not granted a guaranteed time at all where it can finish its Req_i request. (1)
Now, we discuss how PISCOT schedules these requests. First, each core can only issue requests during its dedicated TDM slot (Lemma 5.3). Second, an owner core responds to requests from another core immediately during this other core slot and not its own slot (Lemma 5.5). Accordingly, for our dictated scenario, Req_i has a guaranteed time slot to be issued on the `Request Bus`. Finally, since the `Response Bus` services requests in their order on the `Service Queue`, Req_i is guaranteed to finish its data transfer once all requests in front of it in the `Service Queue` finish their transfers. Now, it remains to show that the number of these requests is bounded. According to the operation described at the beginning of this section, PISCOT only allows a maximum of one request from any core at any time in the `Service Queue`. As a result, Req_i cannot have more than $N - 1$ requests ahead of it `Service Queue`, which guarantees it a bound on the time it can be serviced (Section 6 provides a detailed latency analysis to derive these bounds). (2)
For now, (2) clearly contradicts (1), which completes the proof. □

6 ANALYTICAL WORST-CASE LATENCY

We derive the WCL suffered by any single request to the cache hierarchy that is managed by PISCOT. In doing so, we will use Figure 5, where the GetM(A) from $C2$ is the request under analysis or *rua*. As previously explained, the system in Figure 5 has three cores. As the figure illustrates, upon the arrival of the *rua* at timestamp t, there is a pending request from $C0$ to the same cache line A, which is initially owned by $C1$ in the M state. Generally, from its arrival to the private cache controller buffer until it completely receives the requested data, a request suffers from three different latency components. Namely, it suffers from latency due to arbitration on the request bus, denoted as *ReqBusL*, latency due to arbitration on the response bus, denoted as *ResBusL*, and finally the latency needed to transfer its data from the memory denoted as *AccL*. The *AccL* depends on the time required to access the shared memory and transfer one cache line to the requesting core's private cache. Now, we derive the worst-case latency of each of the other two components. It is worth mentioning that since the architecture and operation of PISCOT is independent of the underlying protocol. By construction, the driven latency bounds are the same for both MSI and MESI. We will further evaluate this in Section 7.6.

LEMMA 6.1. ***Worst-Case Request-Bus Latency*** *($ReqBusL^{WC}$). For a system with N cores, a request has to wait for a maximum of $ReqBusL^{WC}$ cycles as calculated in Equation 1 before it is granted access to the request bus, where S^{Req} is the TDM slot width of the request bus in cycles.*

$$ReqBusL^{WC} = N \cdot S^{Req} \tag{1}$$

Recall that the request bus is managed using a TDM arbiter. In the worst case, the *rua* arrives such that its core has just missed its own slot. Since we have N cores and each core is allocated one TDM slot of width S^{Req}, the *rua* has to wait for $N \cdot S^{Req}$ cycles before its corresponding core gets another slot. In Figure 5, $S^{Req} = 4$ cycles and $N = 3$; thus, the GetM(A) from $C2$ waits until $t + 12$ to gain access to the bus.

LEMMA 6.2. ***Worst-Case Response-Bus Latency*** *($ResBusL^{WC}$). For a system with N cores, a request has to wait for a maximum of $ResBusL^{WC}$ cycles from its arrival time to the `Service Queue` before it can start receiving its requested data. $ResBusL^{WC}$ is calculated by Equation 2, where S^{Res} is the time required to conduct one memory transfer on the response bus.*

$$ResBusL^{WC} = (2 \cdot N - 1) \cdot S^{Res} \tag{2}$$

Recall that the `Response Bus` services requests that arrive to the `Service Queue` from the `Request Bus` in a FCFS fashion. In addition, PISCOT allows each core to have at most one request in the `Service Queue` at any given time. Accordingly, the *rua* waits in worst-case for a request from every other core to get serviced. Moreover, in worst-case, each request can require two memory transfers. This is because each request can be modified by another core and hence requires a write-back before the shared memory can send the updated data to the requesting core. Since we have $N - 1$ other cores, this consumes a total of $(N - 1) \cdot 2 \cdot S^{Res}$. Finally, the *rua* itself in worst-case requires a write-back before it can start transferring its own data, which consumes an additional S^{Res}. This leads to $ResBusL^{WC} = (N - 1) \cdot 2 \cdot S^{Res} + S^{Res}$ or $(2 \cdot N - 1) \cdot S^{Res}$. In Figure 5, where $S^{Res} = 50$ cycles, the GetM(A) from $C2$ incurs a $ResBusL$ from $t + 8$ to $t + 258$, which is 250 cycles.

LEMMA 6.3. ***Total Request Worst-Case Latency*** *($TotL^{WC}$). For a system with N cores, the maximum total latency that a request can encounter from its arrival time to its private cache controller before it can start receiving its requested data can be calculated as:*

$$TotL^{WC} = N \cdot (S^{Req} + 2S^{Res}) \tag{3}$$

Since $TotL^{WC} = ReqBusL^{WC} + RespBusL^{WC} + accL$, the proof directly follows from Lemmas 6.1 and 6.2, and the fact that $accL = S^{Res}$ per definition.

6.1 Direct Cache-to-Cache Communication

In this case, only one response slot is needed for any request as Lemma 6.4 proves. Therefore, the total request WCL for such architecture reduces to the value in Lemma 6.5.

LEMMA 6.4. ***Worst-Case Response-Bus Latency with Cache-to-Cache Support*** *($ResBusL^{WC}_{C2C}$). For a system with N cores that supports direct communication among cores' private caches, the maximum latency a request can suffer from its arrival time to the global response queue before it can start receiving its requested data can be calculated as in Equation 4, where S^{Res} is the time required to conduct a memory transfer on the response bus.*

$$ResBusL^{WC}_{C2C} = (N - 1) \cdot S^{Res} \tag{4}$$

The proof directly follows from the proof of Lemma 6.2, with the exception that only one response slot is required per core instead of two as follows. For any request, there are three possibilities. 1) A core requests to read from or write to a cache line that is up-to-date at the shared memory. In this case, the shared memory transfers this line to the requesting core. 2) A core requests to write to a line that is modified by another core. Thus, the owner

core has to send this line to the requesting core. Since the latter is going to update the line, the shared memory does not need to receive the line at the moment. 3) A core requests to read from a line that is modified by another core. In this case, the owner has to send this line to both the requesting core and the shared memory. However, since the architecture supports cache-to-cache communication, the data can be sent to both at the same slot. This proves that under all these possibilities, only one response slot is needed instead of two compared to Lemma 6.2. In conclusion, the $ResBusL^{WC}_{C2C} = (N-1) \cdot S^{Res}$.

LEMMA 6.5. ***Total Request Worst-Case Latency with Cache-to-Cache Support*** *($TotL^{WC}_{C2C}$). For a system with N cores that supports direct communication among cores' private caches, the maximum total latency that a request can encounter from its arrival time to its private cache controller before it can start receiving its requested data can be calculated as:*

$$TotL^{WC} = N \cdot (S^{Req} + S^{Res}) \tag{5}$$

The proof directly follows from summing the latency components in Lemmas 6.1 and 6.4, and the *AccL*.

6.2 Total Task's Worst-Case Memory Latency

The latencies derived so far are concerned with a single memory request. However, to derive the total task's WCET, the total memory latency, $WCML$, has to be obtained and then added to the worst-case computation time, $WCCT$, such that:

$$WCET = WCCT + WCML \tag{6}$$

Let WCL_{Req} to be the per-request WCL to differentiate it from the total $WCML$, where WCL_{Req} is either the $TotL^{WC}$ in Lemma 6.3 if no cache-to-cache is supported, or the $TotL^{WC}_{C2C}$ in Lemma 6.5 otherwise. We now show different approaches to utilize this WCL_{Req} to derive $WCML$.

6.2.1 Using total number of requests. The first approach directly obtains $WCML$ through Equation 7, where $NReq$ is the worst-case total number of issued memory requests by the task. $NReq$ can be obtained by statically analyzing the task in isolation [15].

$$WCML = NReq \times WCL_{Req} \tag{7}$$

6.2.2 Distinction between private and shared data. Although the bound provided in Equation 7 is safe, it is rather pessimistic. This is because it assumes that all requests are misses, while in reality some of the requests will hit in the private caches and thus suffer a much less latency than WCL_{Req}. One challenge in data-sharing systems is that whether a task access to shared data hits or misses in the private cache depends on the access pattern of competing tasks, entailing that no reasoning can be made about whether this access hits or misses in the private cache by statically analyzing the task in isolation. Even worse, since shared cache lines can conflict with private lines in the core's private cache and hence evict each other, no analysis can be applied to access to private data as well. In this case, Equation 7 applies. In contrast, if private and shared data are isolated from each other; for instance, by mapping them to different cache sets, tighter memory latency bounds can be obtained for requests to the private data. Assuming this isolation, a task's hit ratio to the private data obtained from analyzing the task in isolation still holds upon interference from co-running other tasks. As a result, in such system, we can obtain the $WCML$ as in Equation 8, where $NReq^{priv}$ is the number of requests to private data, among them $NReq^{priv}_{hit}$ are hits in the private cache, and $NReq^{priv}_{miss}$ are misses. L_{hit} is the hit latency of the private cache and $NReq^{shrd}$ is the number of requests to shared data. Since $L_{hit} << WCL_{req}$ (L_{hit} is one or two cycles in modern architectures), the $WCML$ bound in Equation 8 is generally tighter than that of Equation 7. The actual values depend on the ratio of requests to private and shared data, and hence, is application dependent.

$$WCML = NReq^{priv}_{hit} \times L_{hit} + (NReq^{priv}_{miss} + NReq^{shrd}) \times WCL_{Req} \tag{8}$$

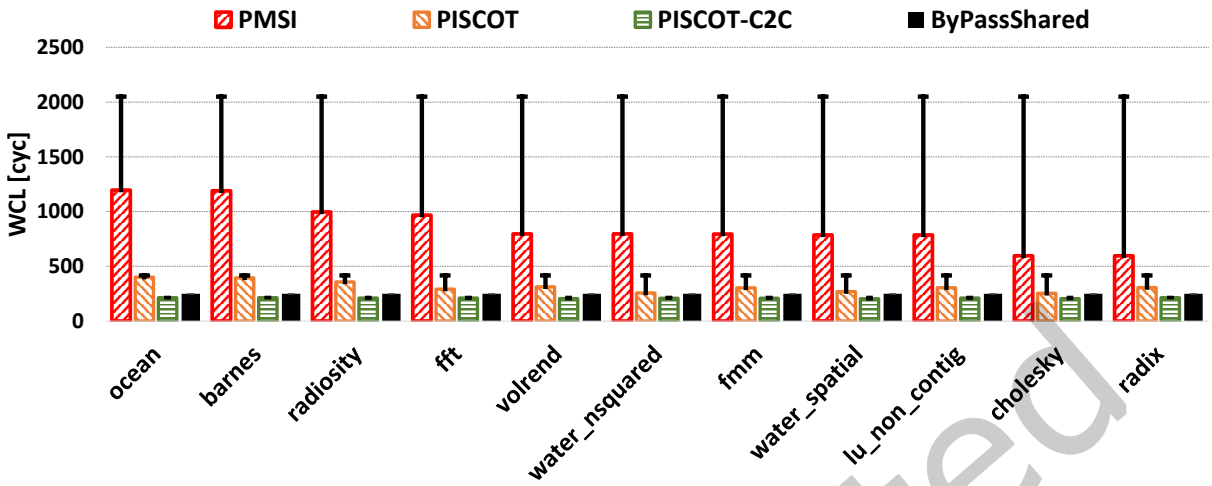


Fig. 6. Per-request WCL for SPLASH-3 suite.

6.3 Replacement of Dirty Cache Lines

The analysis in Lemmas 6.1–6.5 assumes that when a request misses in the private cache, it is sent directly to the bus arbiter to fetch the requested data. However, it is possible that the requested cache line is mapped to an entry that already has a valid data of another cache line. This is called a cache conflict. In this case, the previous cache line is to be evicted from the private cache and the requested cache line is to be fetched to the same entry. If the evicted cache line has modified data, it has to be written first to the shared memory; otherwise, this data is going to be lost. This adds an extra latency of one memory transfer (or S^{Res}) for each miss request in the worst case. In other words, this adds $N \times S^{Res}$ to the latencies in Lemmas 6.3 and 6.5. However, assuming that every request is going to an eviction to a modified line is overly pessimistic and a tighter bound can be obtained as follows.

6.3.1 Total number of writes. Since the additional latency component is caused only upon evicting a dirty cache line, the total number of these replacements is bounded by the total number of write requests of the task, $WReq$. Accordingly, the effect of the replacement is better to be considered at the task level by updating Equation 7 to:

$$WCML = NReq \times WCL_{Req} + WReq \times S^{Res} \tag{9}$$

6.3.2 Distinction between private and shared data. Moreover, if the isolation between private and shared data discussed in Section 6.2.2 is adopted, the delay effects of replacement can be further reduced. This is because the number of replacements happening withing private data can also be obtained from analyzing the task using existing static analysis tools. Therefore, integrating the effect of replacements in Equation 8 leads to the $WCML$ in Equation 10, where $NRepl^{priv}$ is the worst-case number of dirty cache line replacements within private data, $WReq^{shrd}$ is the worst-case number of write requests to shared data.

$$WCML = NReq_{hit}^{priv} \times L_{hit} + NRepl^{priv} \times S^{Res} + (NReq_{miss}^{priv} + NReq^{shrd}) \times WCL_{Req} + WReq^{shrd} \times S^{Res} \tag{10}$$

7 EVALUATION

We develop an open-source simulation framework [2] to evaluate the performance of PISCOT and compare it with state-of-the-art solutions. The simulation environment consists of a multi-core system with configurable number of

[2]https://gitlab.com/FanosLab/piscot

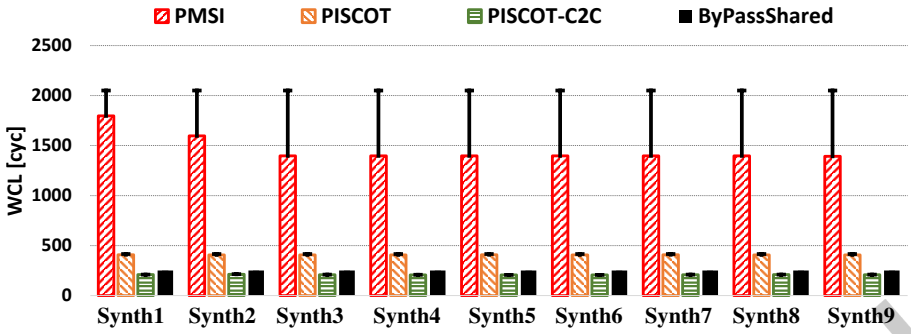


Fig. 7. Per-request WCL for the synthetic workloads.

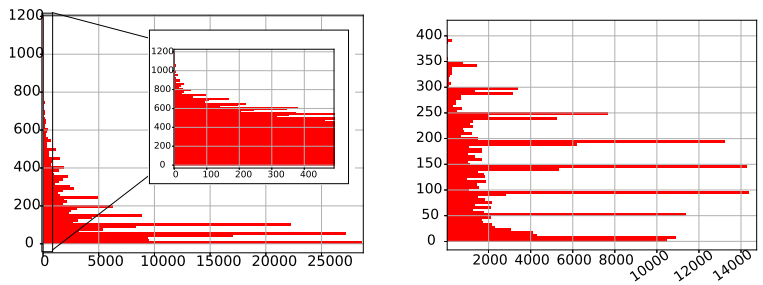
cores and cache organization. The simulator parameters are chosen to emulate the behavior of quad-core system running at 2.5GHz with out-of-order pipelines, 8KB direct-mapped L1 per-core private cache, and a 4MB 8-ways set-associative L2 shared cache across all cores. Both L1 and LLC have a cache line size of 64 bytes. Furthermore, each core and LLC/shared memory are embedded with cache controller units which implement the MSI coherence state machine as described in sSection 2. In addition, we also compare in Section 7.6 both PISCOT and the baseline FCFS using the MESI protocol.

The collection of these coherence controllers are connected to the memory bus using bi-directional FIFO queues which are used for buffering incoming and outgoing messages generated by the coherence protocol. For PISCOT, the request and the response buses are split and operate independently. The former uses work-conserving TDM arbitration amongst cores with a slot width of 4 cycles (S^{Req} = 4 cycles), while the latter services the responses in FCFS fashion assuming the access latency to the LLC is fixed equals to 50 cycles (S^{Res} = 50 cycles). The effect of the LLC access latency on the system performance is explored in Section 7.4. Accesses that hit in the L1 consume one cycle. We use a perfect LLC cache similar to existing works [14, 22] to avoid extra delays from accessing off-chip DRAM to measure only coherence and memory bus latencies. The DRAM access overheads can be computed using other approaches such as [9, 16], and they are additive [44] to the latencies derived in this work.

The address translation between virtual CPU address and physical memory address is disabled such that all memory addresses generated by the cores are physical memory addresses. The maximum number of pending requests ($N_{Pending}$) a core can issue is set to 4 requests. This allows the core to issue multiple memory requests in parallel. Section 7.5 studies the effect of the $N_{Pending}$'s value on the system performance. The private cache controller has to track all pending requests issued on the bus and stall the core pipeline if it reaches to the maximum $N_{Pending}$. In addition, the controller needs to ensure that there is no more than one coherent message issued on the bus or in its local buffer in case of multiple cache misses occur on the same cache line. ***Benchmarks.*** We use the SPLASH-3 benchmark suite since it is a representative of multi-threaded applications with shared data. In addition, we craft 9 synthetic workloads to stress the behavior of the evaluated approaches. All the synthetic workload resemble the maximum data sharing among cores (all lines are shared). They only differ in their memory intensity and the read/write ratio.

7.1 Per-Request Worst-Case Latency

Figures 6 and 7 depict the WCL for any request to the cache hierarchy for both SPLASH-3 benchmarks and the synthetic workloads, respectively. The figures show both the analytical WCL bounds (T bars) and the observed (experimental) WCL (colored solid bars). We compare the WCL of the two PISCOT schemes (where PISCOT-C2C is the one supporting cache-to-cache communication) with PMSI and not caching the shared data (BypassShared)



(a) MSI (b) PISCOT

Fig. 8. Request latency histogram for the Ocean benchmark with the no cache-to-cache transfer architecture. y-axis is the latency in cycles and x-axis is the number of requests encounter this latency.

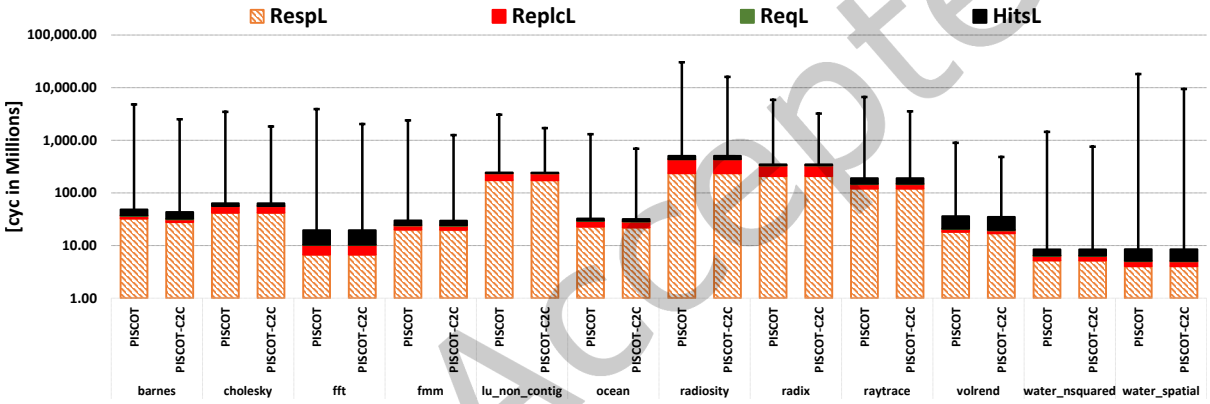


Fig. 9. Total observed and analytical memory latency of Splash-3 benchmarks. Values in y-axis are in log scale.

approaches in [11, 26]. From this experiment, we make the following observations. 1) For both PISCOT and PISCOT-C2C, all the observed WC latencies are always within the analytical worst-case latency bounds. 2) PISCOT shows up to 4.9× improvement in the analytical WCL compared to PMSI. The analytical WCL of PMSI is 2050 cycles compared to 416 and 216 cycles in PISCOT and PISCOT-C2C, respectively. 3) Compared to PMSI, the observed WCLs in PISCOT and PISCOT-C2C achieve up to 2.74× and 4× tighter bounds on average across benchmarks, respectively.

4) PMSI incurs a large gap between experimental and analytical WCLs. In the SPLASH-3 benchmarks (Figure 6), this gap ranges from 70% (barnes and ocean) and reaches up to 3.4× (cholesky and radix). This is because PMSI's analytical WCL assumes a pathological worst-case scenario that is hard to construct in real applications. On the other hand, PISCOT achieves a tighter bound for the derived WCL. PISCOT achieves this tightness by enforcing FCFS arbitration policy on the response bus.

To further investigate the behavior of PISCOT and conventional split-transaction MSI, Figure 8 plots the observed latencies for requests for one of the BMs (Ocean) (others show similar behavior) for both solutions. As the figure illustrates, for MSI 8a, it shows a huge latency variability. Although most of the requests finish relatively fast, there are requests that their latencies reach up to 1200 cycles. On the other hand, PISCOT encounters less variability

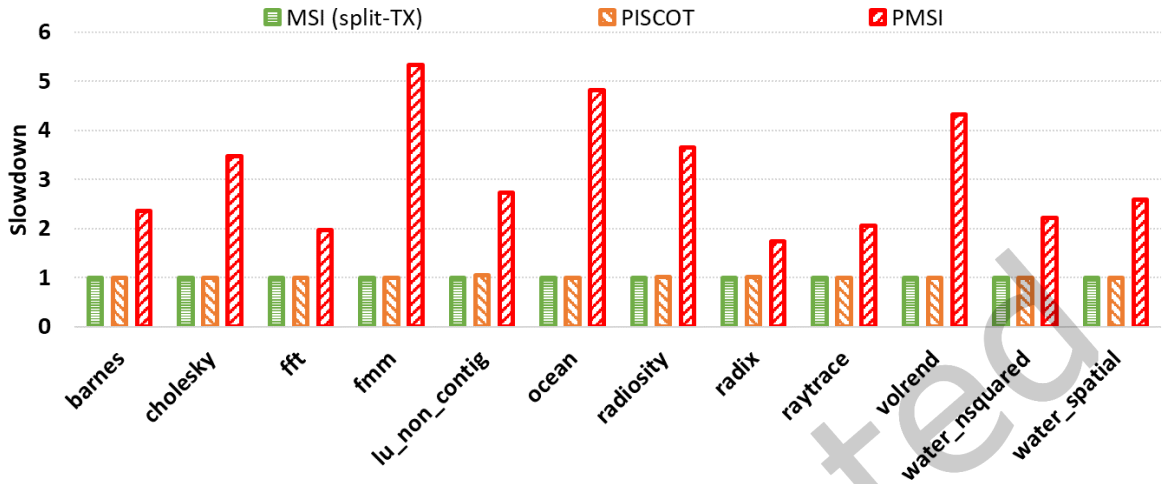


Fig. 10. Execution time slowdown compared to conventional MSI protocol with FCFS split-transaction bus.

(Requests are suffering between $0 - 400$ cycles and all latencies under PISCOT operation are lower than its corresponding analytical bounds, which confirms the predictability of PISCOT.

7.2 Total Worst Case Latency

In this experiment we are interested in calculating the total memory WCL suffered by the total number of memory requests generated by a core during a period of time t. Figure 9 shows both the analytical bound for the total WCL derived by Equation 7 (T bars) and the observed total latencies (colored solid bars). Furthermore, the observed one is decomposed to its sub-components: a) the request bus arbitration latency, b) the response bus memory transfer latency, c) the hit latency in the core's private cache, and d) the write-backs latency due to replacement. From Figure 9, we conclude the following observations. 1) The response bus latency component dominates the total WCL for all applications. For instance, the total observed response latency reach up to 8× (barnes and volrend) and 4.3× on average larger than the replacement latency. This emphasises the conclusion we made in Section 6.3 that the effect of the eviction delays should be considered at the task-level and not the request-level. 2) Since SPLASH-3 applications exhibit a reduced ratio of writes compared to reads, they do not stress the difference between PISCOT and PISCOT-C2C in the observed response bus latency. Therefore, to further show this effect, we execute synthetic experiments using the synthetic benchmarks that are used to generate WCL in Figure 7 except that we change the percentage of the CPU memory write request to 50% of total memory requests. The results show that PISCOT-C2C achieves up to 1.74× (1.56× on average) higher bandwidth compared to PISCOT.

7.3 Average-Case Performance

Figure 10 shows the slowdown of PISCOT and PMSI compared to the conventional MSI with split-transaction FCFS bus. PMSI's slowdown is 2× on average (and up to 4.3×) across all benchmarks. This is due to the coupling of coherence and data transfer on the same TDM bus as explained in Section 4 in addition to the enforced protocol changes. Authors of [14] compared PMSI with an MSI+conventional TDM arbiter, for which they reported that PMSI showed only a 45% slowdown. Recall here we consider MSI+split-transaction bus. These results combined

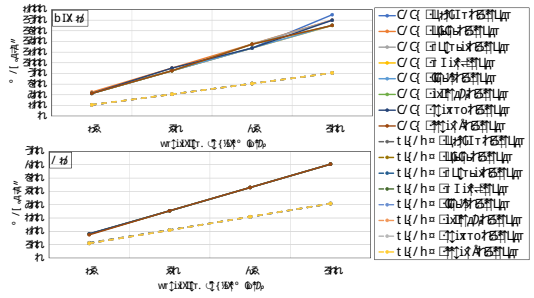


Fig. 11. Effect of shared cache access latency on predictability

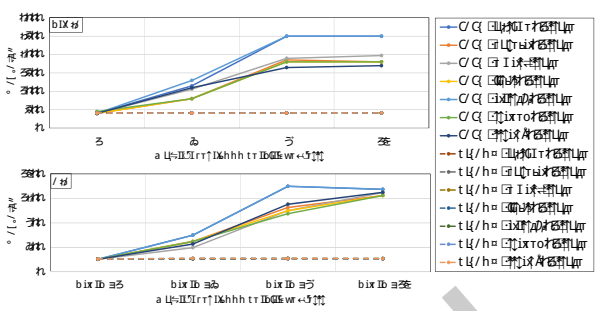


Fig. 12. Effect of OOO depth on predictability

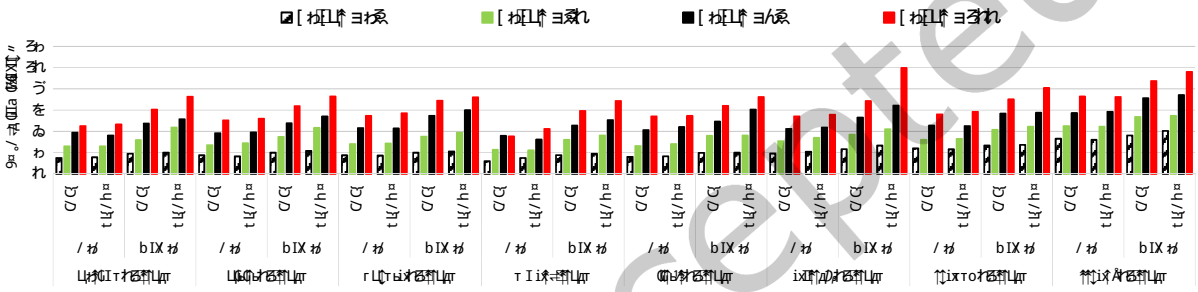


Fig. 13. Effect of shared cache access latency on execution time

emphasise our observation that the split-bus architecture can significantly increase performance compared to the traditionally considered bus architectures by the real-time community. On the other hand, Figure 10 shows that PISCOT achieves comparable results with slowdown in the range of $1\% - 4\%$. This is clearly a negligible cost for achieving timing predictability with tight latency bounds.

We also observe that PISCOT improves the bandwidth utilization for the SPLASH-3 benchmarks by 12× compared to PMSI (results are not shown due to space limitation). This significant improvement is because PMSI adopting the traditional TDM, which wastes many bus slots in only issuing coherence requests as we detailed in Section 4. On the other hand, PISCOT maximizes bus utilization by splitting the coherence and response into two buses with two different slot widths and arbitration.

7.4 Effect of Shared Cache Access Latency

Effect on Predictability. The analysis in Section 6 (e.g. Equations 3 and 5) shows a linear relationship between the worst-case latency and the ResponseBus's slot width, S^{Res}. The latter is defined in Section 6 as the time required to conduct a memory transfer from LLC to any of the cores on the response bus. To study the effect of this time on the system's predictability and performance, we sweep S^{Resp} to have the values of 25, 50, 75, or 100 cycles. Figure 11 delineates the observed per-request WCL both for PISCOT and baseline FCFS with no cache-to-cache interconnect (upper, NoC2C) and with cache-to-cache interconnect (lower, C2C) using the EEMBC benchmarks with the aforementioned setup of a quad-core system. 1) From the figure, it is clear that the slope of PISCOT is smaller than that of FCFS for all benchmarks both in case of NoC2C and C2C. This shows that the predictability that PISCOT offers over the baseline commodity FCFS gets more crucial for higher access

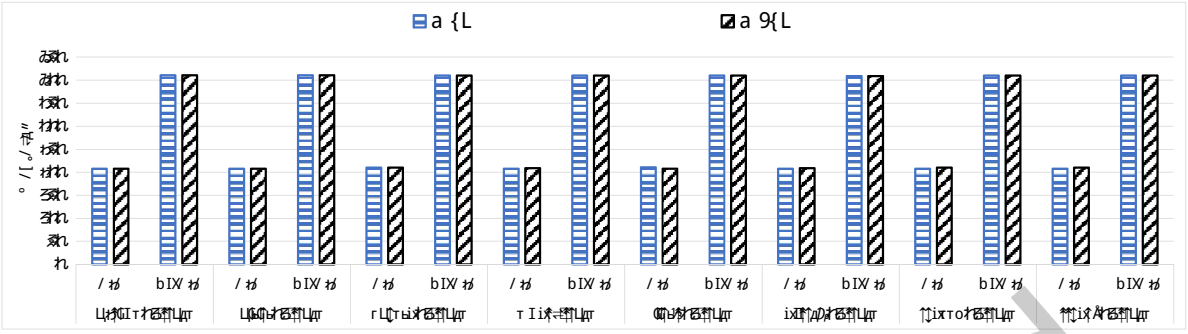


Fig. 14. PISCOT's WCL for both MSI and MESI

latency of the shared cache. 2) Across all the values of the ResponseBus's slot width, the observed WCL for PISCOT is less than the analytical bounds dictated in Section 6. Using Equations 3 and 5, the analytical bounds are 116, 216, 316, 416 (216, 416, 616, 816) for C2C (NoC2C) with a slot width of 25, 50, 75, and 100, respectively. This confirms the safeness of the provided bounds. Furthermore, all observed WCLs are very close to the derived bounds, which also confirms the tightness of the conducted analysis thanks to PISCOT's predictable-by-design architecture. 3) Finally, comparing both C2C and NoC2C results across different slot widths show the importance of having the hardware capability of supporting direct cache-to-cache transfers amond cores without the need to go through the LLC in improving WCL.

Effect on Performance. The improved predictability of PISCOT compared to commodity arbiters comes at a slight performance cost. Figure 13 shows the execution time for PISCOT vs baseline FCFS assuming both NoC2C and C2C architectures with the same EEMBC benchmark suite quad-core setup as before. The results show that increasing the LLC access time (ResponseBus's slot width), PISCOT's performance degrades slightly compared to baseline FCFS. In case of C2C, this degradation in average ranges between 3% to 5% for different slot widths. On the other hand, in the case of NoC2C, this degradation becomes worse and ranges from 7% in case of $S^{Resp} = 25$ cycles to 17% in case of $S^{Resp} = 100$. The fundamental reason for this degradation is the design choice of PISCOT to reduce the latency bounds by allowing each core to have only one request in-service at any given time as detailed in Section 5.

7.5 Effect of Out-of-Order Depth

One important aspect of PISCOT compared to all previous work in predictable coherence solutions is to support OOO cores. On the other hand, to ensure tight latency bounds, PISCOT while allowing cores to have multiple pending requests, it limits the number of in-service requests from any core to one. In this section, we study the effect of the maximum number of possible pending requests on predictability, $N_{pending}$. This number usually is dictated by the number of entries in MSHR registers. We sweep this number from 1 to 16 and we plot the WCLs in Figure 12. The setup is exactly as before using a quad-core system. As Figure 12 illustrates, the WCL in all experiments is fixed and independent of the value of $N_{pending}$ in case of PISCOT. On the other hand, the observed WCL in case of commodity FCFS increases with the increase of $N_{pending}$ since a request will suffer more interference by increasing the number of possible requests in the service. For some benchmarks, this increase saturates after a certain value of $N_{pending}$. Intuitively, this is because for those specific experiments, some benchmarks do not have enough parallelism to saturate the possible number of pending requests.

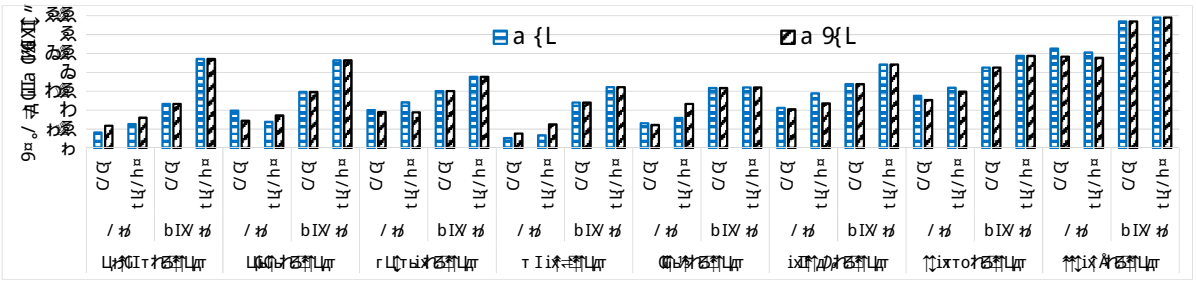


Fig. 15. MESI vs MSI: EEMBC execution time for PISCOT and FCFS with cache-to-cache and no -cache-to-cache architectures

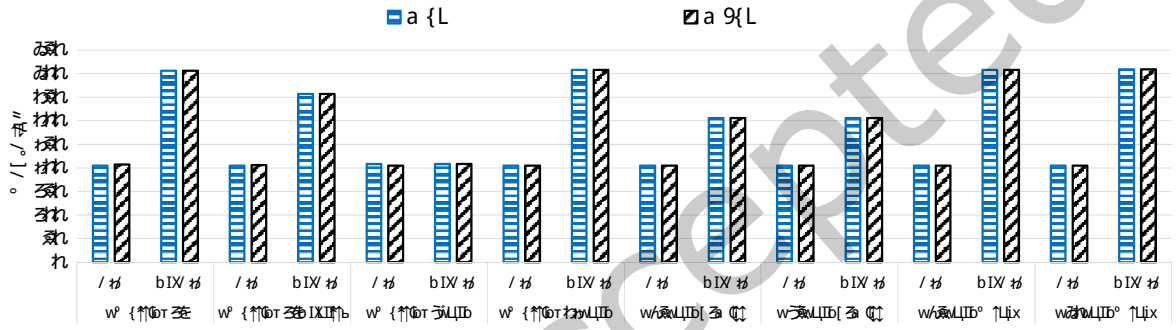


Fig. 16. MESI vs MSI: WCL

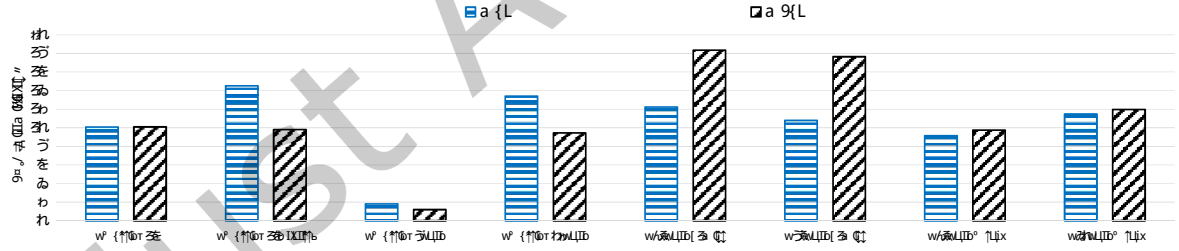


Fig. 17. MESI vs MSI: execution time

7.6 MSI vs MESI

A fundamental contribution of PISCOT over existing predictable cache coherence solutions is that it enables predictability without the need to modify the underlying hardware cache coherence protocols. To show this advantage, we implement the Modified-Exclusive-Shared-Invalid (MESI) protocol and integrate it in our simulation framework that has PISCOT as its bus arbiter. This integration does not require any changes to the PISCOT architecture. We only enabled the use of MESI as the underlying coherence protocol in the cache controllers and PISCOT seemingly works with it. Since we release the source code of our simulation infrastructure, one can add any other protocol of choice. Figure 14 illustrates the observed WCL for PISCOT using both MSI and MESI

Table 4. Synthetic micro-benchmarks description

Benchmark	Description
RWStride16	Sequential read-modify-write memory accesses with stride offset equals 16 Bytes.
RWStride16NoIntrf	Sequential read-modify-write memory accesses with stride offset equals 16 Bytes, and no shared data between cores
RWStride8Rand	Random read-modify-write memory accesses with stride offset equals 8 Bytes.
RWStride32Rand	Random read-modify-write memory accesses with stride offset equals 32 Bytes.
R75RandL1Miss	Random memory accesses with read percentage equals 75% and all requests are miss in L1 Cache.
R85RandL1Miss	Random memory accesses with read percentage equals 85% and all requests are miss in L1 Cache.
R75RandWrap	Random memory accesses with read percentage equals 75% and address wrapping every 1024 KBytes.
R40RandWrap	Random memory accesses with read percentage equals 40% and address wrapping every 1024 KBytes.

protocols for the EEMBC quad-core setup. As results emphasis, PISCOT's WCL is the same independent of the protocol, and less than (but very close) to the analytical bounds ensuring safe yet tight bounds. Figure 15, on the other hand, delineates the execution time for both protocols using PISCOT and baseline FCFS. The results show that across the EEMBC setup, there is very little difference (within 1%) in behavior between MSI and MESI. We believe this is due to the nature of the EEMBC setup that maximizes data sharing by running the exact same benchmark across all the four cores. To further investigate the differences between MESI and MSI behavior, we create a set of synthetic micro-benchmarks (Table 4) that stresses the benefits and drawbacks of both protocols. The WCL and execution time results are depicted in Figures 16 and 17, respectively. Figure 16 confirms again that the derived latency bounds for PISCOT hold for both MSI and MESI protocols. Results in the figures are for the C2C scenarios. Results in Figure 17 sheds light on the differences between MESI and MSI protocols. MESI achieves considerable improvement over MSI for the read-modify-write micro-benchmarks: 48%, 52%, and 42% for the RWStride16NoIntrf, RWStride8Rand, and RWStride32Rand, respectively. On the other hand, MSI outperforms MESI for the read-dominant random benchmarks with up to 38% better performance for the R85RandL1Miss micro-benchmark. The rationale behind these results is that for the former micro-benchmarks of the read-modify-write and high locality behavior, the MESI protocol leverages the E state to move silently (and faster) to the M state. This emphasises the benefit of the E state in decreasing memory traffic on the bus in case of a write after a read. In contrast, for the latter benchmarks with the random and read-dominant pattern, MESI does not exploit the same benefit from the E state. Contrarily, with evictions introduced by the random behavior, MESI has to write-back blocks in the E state to the main memory (even if they are not modified), while MSI does not need to write-back those blocks if they are in the S state.

7.7 Effect of Number of Contending Cores on Latency Bounds

Increasing the number of cores competing to access shared memory, the contention effects from these cores will increase. This is clear from the latency bound equations both for PMSI in Section 4.3 as well as PISCOT's analysis in Section 6, where these bounds are functions in the number of cores. In Figure 18, we delineate the latency bounds for different number of cores to see how different predictable solutions address the increasing contention. Contention (latency bounds) in PMSI increases the most. This confirms its analysis since its bounds are quadratic in the number of cores. On the other hand, PISCOT with its two versions has a linear relationship between the bounds and the number of cores. Hence, it shows better control of the increasing contention when the number of cores increase in the system.

8 CONCLUSION

PISCOT is a predictable and coherent bus architecture that provides significantly tighter bounds than existing predictable coherence protocols with a performance near to that achieved by conventional high-performance arbiters. PISCOT achieves this by decoupling the data responses from their coherence requests utilizing a split-transaction

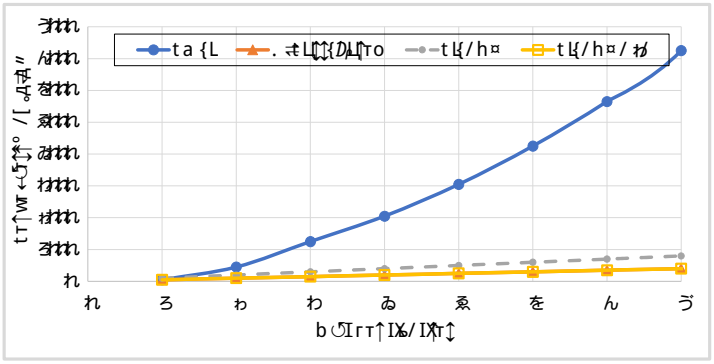


Fig. 18. Latency bounds for different number of contending cores.

predictable bus arbiter. PISCOT can be realized without any modifications to the coherence protocol or cache controller. In this work, PISCOT was integrated to both MSI and MESI protocols. We have also studied the effects of different architectural parameters on the performance of PISCOT including LLC access latency as well as number of outstanding requests from OOO cores. Results show that PISCOTachieves 4× tighter memory latency bounds for a quad-core system with 5× (2.8× on average) better performance compared to the state-of-the-art predictable coherence solutions.

REFERENCES

[1] WL Bain Jr and SR Ahuja. 1981. Performance analysis of high-speed digital buses for multiprocessing systems. In *Proceedings of the 8th annual symposium on Computer Architecture*. 107–133.

[2] Ayoosh Bansal, Jayati Singh, Yifan Hao, Jen-Yang Wen, Renato Mancuso, and Marco Caccamo. 2019. Cache Where you Want! Reconciling Predictability and Coherent Caching. *arXiv preprint arXiv:1909.05349* (2019).

[3] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. 2016. Contention-free execution of automotive applications on a clustered many-core platform. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*.

[4] M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, and F. D. Smith. 2016. Reconciling the Tension Between Hardware Isolation and Data Sharing in Mixed-Criticality, Multicore Systems. In *IEEE Real-Time Systems Symposium (RTSS)*.

[5] Michael A Fischer. 1988. Fair arbitration technique for a split transaction bus in a multiprocessor computer system. US Patent 4,785,394.

[6] Freescale semicondutor. 2016. QorIQ T2080 Reference Manual. Also supports T2081. Document Number: T2080RM. Rev. 3, 11/2016.

[7] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. 2015. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Comput. Surv.* (2015).

[8] Giovani Gracioli and Antônio Augusto Fröhlich. 2015. On the Design and Evaluation of a Real-Time Operating System for Cache-Coherent Multicore Architectures. *ACM SIGOPS Oper. Syst. Rev.* (2015).

[9] Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. 2018. A comparative study of predictable dram controllers. *ACM Transactions on Embedded Computing Systems (TECS)* (2018).

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

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

[12] Mohamed Hassan. 2018. On the Off-chip Memory Latency of Real-Time Systems: Is DDR DRAM Really the Best Option?. In *IEEE Real-Time Systems Symposium (RTSS)*.

[13] Mohamed Hassan. 2020. Discriminative Coherence: Balancing Performance and Latency Bounds in Data-sharing Multi-Core Real-Time Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 1–22.

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

[15] M. Hassan and H. Patel. 2016. Criticality- and Requirement-Aware Bus Arbitration for Multi-Core Mixed Criticality Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
[16] Mohamed Hassan and Rodolfo Pellizzoni. 2018. Bounding DRAM interference in COTS heterogeneous MPSoCs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).
[17] Farouk Hebbache, Mathieu Jan, Florian Brandner, and Laurent Pautet. 2018. Shedding the Shackles of Time-Division Multiplexing. In *IEEE Real-Time Systems Symposium (RTSS)*.
[18] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.
[19] Salah Hessien and Mohamed Hassan. 2020. 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)*. IEEE, 218–230.
[20] Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. 2020. Designing predictable cache coherence protocols for multi-core real-time systems. *IEEE Trans. Comput.* (2020).
[21] Anirudh M. Kaushik and Hiren Patel. 2021. A Systematic Approach to Achieving Tight Worst-Case Latency and High-Performance Under Predictable Cache Coherence. In *proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 1–12.
[22] Anirudh M. Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren Patel. 2019. CARP: A Data Communication Mechanism for Multi-Core Mixed-Criticality Systems. In *IEEE Real-Time Systems Symposium (RTSS)*.
[23] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2011. Bus-aware multicore WCET analysis through TDMA offset bounds. In *Euromicro Conference on Real-Time Systems (ECRTS)*.
[24] Manpreet S Khaira. 1996. Fast first-come first served arbitration method. US Patent 5,574,867.
[25] Namhoon Kim, Micaiah Chisholm, Nathan Otterness, James H Anderson, and F Donelson Smith. 2017. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
[26] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. 2010. Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures.. In *International Conference on Real-Time and Network Systems*.
[27] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. 2013. Real-time cache management framework for multi-core architectures. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
[28] MILO MK MARTIN, MARK D HILL, and DANIEL J SORIN. 2012. Why On-Chip Cache Coherence Is Here to Stay. *Communications of ACM* (2012).
[29] Marco Paolieri, Eduardo Quiñones, Francisco J Cazorla, Guillem Bernat, and Mateo Valero. 2009. Hardware support for WCET analysis of hard real-time multicore systems. *ACM SIGARCH Computer Architecture News* (2009).
[30] Rodolfo Pellizzoni, Bach D Bui, Marco Caccamo, and Lui Sha. 2008. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *IEEE Real-Time Systems Symposium (RTSS)*.
[31] Francesco Poletti, Davide Bertozzi, Luca Benini, and Alessandro Bogliolo. 2003. Performance analysis of arbitration policies for SoC communication architectures. *Design Automation for Embedded Systems* (2003).
[32] Fong Pong and Michel Dubois. 1995. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems* (1995).
[33] Roger Pujol, Hamid Tabani, Jaume Abella, Mohamed Hassan, and Francisco J. Cazorla. 2020. Empirical Evidence for MPSoCs in Critical Systems: The Case of NXP's T2080 Cache Coherence. In *IEEE Design Automation and Test in Europe (DATE)*. 1–4.
[34] D. Radack et al. (Rockwell Collins). 2018. Civil Certification of Multi-core Processing Systems in Commercial Avionics.
[35] Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. 2009. Towards time-predictable data caches for chip-multiprocessors. In *Springer International Workshop on Software Technolgies for Embedded and Ubiquitous Systems (IFIP)*.
[36] Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. 2019. Modeling cache coherence to expose interference. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[37] Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. 2020. On How to Identify Cache Coherence: Case of the NXP QorIQ T4240. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
[38] Ashok Singhal, Bjorn Liencres, Jeff Price, Frederick M Cerauskis, David Broniarczyk, Gerald Cheung, Erik Hagersten, and Nalini Agarwal. 1999. Implementing snooping on a split-transaction computer system bus. US Patent 5,978,874.
[39] Daniel J Sorin, Mark D Hill, and David A Wood. 2011. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture* (2011).
[40] N. Sritharan, A. M. Kaushik, M. Hassan, and H. Patel. 2017. Hourglass: Predictable time-based cache coherence protocol for dual-critical multi-core systems. (2017).
[41] Nivedita Sritharan, Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. 2019. Enabling Predictable, Simultaneous and Coherent Data Sharing in Mixed Criticality Systems. (2019).

[42] Man-Ki Yoon, Jung-Eun Kim, and Lui Sha. 2011. Optimizing tunable WCET with shared resource allocation and arbitration in hard real-time multicore systems. In *IEEE Real-Time Systems Symposium (RTSS)*.
[43] Mohamed Younis and Mohamed Aboutabl. 2002. Communication handling in integrated modular avionics. US Patent App. 09/821,601.
[44] Heechul Yun, Rodolfo Pellizzoni, and Prathap Kumar Valsan. 2015. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*.
[45] Dimitrios Ziakas, Allen Baum, Robert A Maddox, and Robert J Safranek. 2010. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *IEEE Symposium on High Performance Interconnects*.