DISCO: Time-Compositional Cache Coherence for Multi-core Real-Time Embedded Systems

Mohamed Hassan, *Member, IEEE*

Abstract—Tasks in modern embedded systems share data and communicate among each other. Nonetheless, the majority of research in real-time systems either assumes that tasks do not share data or prohibits data sharing by design. Only recently, some works investigated solutions to address this limitation and enable data sharing. However, we find these works to suffer from severe limitations. In particular, proposed predictable cache coherence protocols increase the worst-case memory latency (WCL) quadratically due to coherence interference and breaks compositionality by coupling the design and timing analysis of the coherence with the underlying bus arbitration policy. In this paper, we argue that a protocol that distinguishes between non-modifying (read) and modifying (write) memory accesses is key towards reducing the effects of coherence interference on WCL. Accordingly, we propose DISCO, a discriminative coherence solution that capitalizes on this observation to 1) balance average-case performance and WCL, and 2) more importantly, achieves compositionality in the existing of coherence by enabling the decomposition of the effects from coherence and arbitration components. DISCO achieves 7.2× lower latency bounds compared to the state-of-the-art predictable coherence protocol. DISCO also achieves up to 11.4× (5.3× on average) better performance than private cache bypassing for the SPLASH-3 benchmarks.

Index Terms—Cache Coherence, Real-Time Systems, Timing Behavior, Time Compositionality, Embedded Systems

1 Introduction

Unlike the general trend in general-purpose computing systems, whose main focus is solely optimizing the average performance, real-time systems are also mandated to achieve timing *predictability* by guaranteeing that each task's worst-case execution time (WCET) does not exceed a designated deadline. Moreover, with the increasing complexity of multi-core real-time systems, it is often desirable to achieve timing *compositionality* such that the system can be decomposed into components and the timing behaviour of the whole system can be feasibly inferred from the timing analysis of these components [1]. Compositionality, therefore, facilitates the reasoning about the system's timing properties; and hence, enables system's certification, which is key for domains such as automotive and avionics [2]. Compositionality also enables the adoption of incremental design by allowing the integration of new components without the need to re-analyze (and hence, recertify) the old components. Incremental design is widely used in avionics through integrated modular avionics (IMA) [3].

Despite this large volume of research towards proposing predictable system components for real-time systems, one demand from the aforementioned applications is yet to be efficiently addressed: allowing seamless, predictable, and high performance data sharing among different running tasks. Unfortunately, most prior works in real-time systems do not meet this demand. They either assume tasks are not sharing data or prohibit data sharing by design [4]. This is mainly because data sharing is problematic and can lead to significant interference delays [5], [6] or even unpredictable behaviors [7] if it is not carefully addressed.

On the other hand, researchers have already realized the importance of enabling data sharing in the context of real-time systems [4], [6], [7], [8]. The common approach followed by these works is to allow data to be shared among tasks but prevent tasks from simultaneously accessing this shared data in an attempt to accommodate for the data sharing demand, while ensuring system predictability. As a result, large interference delays due to this simultaneous access are avoided altogether. This is achieved by either modifying the task-to-core mapping [4], data-aware scheduling [6], [8], or bypassing caches [4], [5]. The main drawback of such approach is that by disallowing simultaneous access to shared data, it can severely deteriorate the system performance. To improve system performance and enable simultaneous access to shared data, [7], [9], [10] propose predictable cache coherence protocols. The problem with these protocols is that 1) they require complex changes to cache controllers and coherence protocols, which are well-known to be hard-to-verify [11]; 2) more importantly, they result in a significant increase in the worst-case latency (WCL) upon accessing memory due to coherence interference; and 3) they couple coherence with the specifications of the underlying interconnect arbitration scheme, and therefore, the derived bounds are not applicable outside the assumed arbiters in these works. This hinders time-compositionality since the system components of bus arbiters and coherence protocols cannot be designed or analyzed independently.

In this paper, we propose DISCO: a discriminative coherence solution that addresses the aforementioned drawbacks. Towards this target, in our preliminary step of this work [12], we exhaustively studied all possible access scenarios under coherence protocols to distill the main sources of their large coherence delays. As a result of our study we make the following important observation. Significant coherence interference delays that arise from the worst-case scenarios are exclusively due to cache lines being modified in the private caches without an immediate update to the shared cache. The other key observation that DISCO is based on is that the number of memory writes usually represents a small percentage of the total memory requests of applications. We discuss these two observations in details in Section 5. Based on these two observations, we proposed DISCO to prohibit the

• M. Hassan is with the Department of Electrical and Computer Engineering, McMaster University, ON, Canada.
E-mail: mohamed.hassan@mcmaster.ca

caching of modified data in the cores' private caches. All data modifications are carried out at the shared cache. In other words, DISCO intentionally discriminates against write memory requests by forcing them to access the shared cache even if data already exists in the requesting core's private cache. In contrast, read requests are allowed to hit in private caches if their data exists; therefore, reads are managed exactly as in traditional coherence approaches deployed in commodity systems. Since all writes are treated equally, we call this version of the proposed solution, DISCO-AllW. To further improve the system performance, we also proposed another version of our solution that we call DISCO-SharedW. DISCO-SharedW leverages information about tasks' data (namely, whether data is shared or private). DISCO-SharedW relaxes the constraint of DISCO-AllW by allowing private data to be modified in the private caches. It allows both read and write hits to the private data that is not shared among tasks since it causes no coherence interference. Both DISCO-AllW and DISCO-SharedW can be either implemented as a hardware cache coherence protocol (Section 6) or realized in commodity platforms using already available support on these platforms as we discuss in Section 6.4.

However, the preliminary version of this work in [12] faces two limitations. 1) It only assumed a multi-core system with time division multiplexing (TDM) arbitration, and hence, the analysis is also only applicable to TDM-based buses. 2) It only considered the simple Modified-Shared-Invalid (MSI) protocol. To address these limitations, this paper makes the following contributions. 1) We show that one of the main advantages of DISCO compared to the existing works is that it achieves time-compositionality by decoupling the timing behavior of cache coherence and bus arbitration. DISCO achieves predictable and coherent data sharing that independent of the details of the underlying arbitration scheme. Thus, the delay components of each can be added in a compositional fashion [1] as we discuss in Section 7. 2) To emphasis this advantage, we show the operation of DISCO with a wide set of bus arbiters. Namely, in addition to the TDM arbitration covered in [12], we consider: Round Robin (RR), First-Come First-Serve (FCFS), weighted RR (WRR), and Harmonic schedulers. We also present the timing analysis these arbiters and for DISCO-AllW and DISCO-SharedW (Sections 6.3 and 7). It is important to note that these arbiters are shown as an example. The analysis in Section 7 abstracts the latency terms resulting from arbitration such that the WCL of any new arbiter can be directly plugged into the analysis thanks to composability. 3) In addition, we extend the approach at the cache coherence protocol level by applying it to the Modified-Exclusive-Shared-Invalid (MESI) protocol. MESI is adopted by several commercial-of-the-shelf (COTS) multi-core platforms targeting embedded systems. An example is the NXP's T4080 [13] with its four E6500 cores, which found an adoption in the avionics domain [14]. Another example is the ARM's advanced micro controller bus architecture (AMBA) with its coherent hub interface (CHI). 4) We conduct extensive experiments to evaluate each of the proposed approaches, arbiters, and coherence protocols. We also compare against two state-of-the-art competitive solutions: PMSI coherence protocol [7] and cache bypassing [5], [15]. Our evaluation uses both the SPLASH-3 [16] parallel data-sharing benchmarks as well as synthetic experiments that are based on the EEMBC-Auto benchmarks [17]. Results in Section 8 show the notorious improvements that DISCO-AllW and DISCO-SharedW achieve compared to both PMSI and cache bypassing. We summarize these results in Table 1. Compared to [12], experiments to evaluate DISCO using the MESI protocol as well as an extensive study of DISCO integration with different arbiters are included.

TABLE 1: Summary of DISCO improvements over state-of-the-art competitive approaches.

	Per-request WCL		Total WCL				Avg. Performance			
	PMSI	Bypass	PMSI		Bypass		PMSI		Bypass	
	analytical		up to	avg.	up to	avg.	up to	avg.	up to	avg.
DISCO-AllW	7.2×	same	3.3×	2×	65%	42%	100%	12%	2.8×	1.5×
DISCO-SharedW	7.2×	same	6×	3.5×	3.8×	1.5×	3.2×	1.6×	11.4×	5.3×

2 Related Work

Recently, researchers recognized the importance of data sharing in multi-core real-time systems and proposed solutions to handle it [4], [5], [6], [7], [8], [9], [10]. We classify these works into three groups according to their research direction: 1) data-aware scheduling, 2) cache bypassing, and 3) cache coherence protocols.

1) **Data-aware scheduling**. The first direction incorporates data-awareness in the task scheduling to avoid data interference. This is achieved by one of the following means: 1.1) scheduling tasks with shared data such that they never run in parallel [8]; hence, they do not compete for shared data; 1.2) assigning tasks with shared data to the same core [4]; hence, they share the same private cache(s) and do not suffer coherence interference from each other; or 1.3) incorporating run-time performance metrics collected through hardware counters to make data-wise scheduling decisions that mitigate the data sharing effects [6]. This direction enforces new constraints on the system scheduler interference, which deteriorates system schedulability [9]. Unlike these solutions, DISCO does not require any modifications to the system scheduler and coherently handles data sharing in hardware.

2) **Cache bypassing**. A second alternative is cache bypassing, which was first utilized in the context of reducing the shared cache conflict interference [18] but is then used to avoid coherence interference of shared data [4], [5]. If private caches are bypassed, coherence interference is eliminated, but at the expense of degrading average-case performance.

3) **Cache coherence**. The third direction is to make data sharing transparent to the application and the scheduler by handling it completely in hardware either by proposing predictable cache coherence protocols [7], [9], [10], [19], [20] or analyzing coherence protocols in existing commercial-of-the-shelf (COTS) embedded platforms [21], [22]. Cache coherence is notoriously the main solution adopted by COTS multi-core architectures [23]. It has the advantage of enabling data sharing without imposing any restrictions on the real-time scheduler compared to data-aware scheduling solutions. It is also shown to provide high average-case performance compared to both data-aware scheduling and cache bypassing [7], [9]. On the other hand, it suffers a notably high worst-case memory latency due to the introduced coherence interference. For instance, PMSI [7] and CARP [10] have a worst-case latency that is quadratic in the number of cores in the system. We discuss both cache bypassing and cache coherence in more details in Section 5 since they are the most related to this work.

3 System Model

We assume a multi-core architecture, where tasks running on this architecture can share data. The proposed solution does not depend on the core architecture and can be seamlessly deployed for in-order or out-of-order cores.

Memory Hierarchy. Each core has its own private cache(s) and all cores share a last-level cache (LLC) accessed through a shared bus. Data coherence is maintained using a hardware cache coherence protocol. Unlike the preliminary version [12] that focuses only on the MSI protocol, we show that DISCO is independent of the coherence protocol details and hence can work with other more sophisticated protocols. To exemplify, we show the operation of DISCO with the MESI protocol. Cores can also share an off-chip memory, whose interference can be bounded using existing solutions orthogonal to this work [24]. Since the focus of the paper is on cache coherence interference, we avoid DRAM interference by assuming a perfect LLC such that all requests hit into the LLC and do not visit the DRAM. This is necessary to asses the validity of the analytically driven bounds in Section 7. Our bounds and DRAM latency bounds driven by the aforementioned orthogonal works can then be used to compose the total WCETs of tasks [25]. Moreover, such assumption also follows prior related works (e.g. [10], [20], [21]).

Bus Arbitration. Requests that miss in the private cache has to be fetched from the LLC through a shared bus. Since multiple requests from different cores can simultaneously require access to the shared bus, a bus arbiter is a must to resolve this contention. Compared to [12], which focuses solely on TDM arbitration, we cover a wide variety of arbiters including RR, WRR, FCFS, and harmonic arbitration. For all arbiters, we assume that once a request is granted access to the bus by the arbiter, it consumes LLC^{acc} to finish accessing the LLC and transfer the data on the bus to the requesting core.

Task Scheduling. We do not make any assumption on how the executing tasks are scheduled on cores. The proposed approach is orthogonal to task scheduling and should operate in tandem with any schedule.

4 Cache Coherence Background

When multiple cores accessing the same data, the system has to maintain data correctness. Data correctness is achieved when all cores have access to the most up-to-date data. On the other hand, data incorrectness occurs when a core accesses a stale data that has been already changed in another location in the system (e.g. another core's cache). Modern multi-core systems deploy cache coherence protocols to prevent such situation and preserve data correctness. MSI is considered the baseline coherence protocol [11], where many of the commercial-off-the-shelf architectures adopt protocols that inherit its three fundamental states: Modified (M), Shared (S), and Invalid (I) such as the MESIF protocol deployed in Intel's i7 and the MOESI protocol deployed in AMD's Opteron. Therefore, we use it as a mean to explain the basics of a coherence protocol.

If a cache line does not exist in the private cache or its data is stale, its state will be I. The S state indicates that the data of this cache line is valid and is not modified, while the M state indicates that the data of this cache line is valid and modified. Therefore, multiple cores can share a cache line in their private cache in the S state, while only one core can have a cache line in the M state. All other cores in this case will have this line in the I state. If a core has a load/read request to a cache line in the I state, the private cache controller of this core (or for simplicity we refer to this throughout the paper as just the core) issues a $GetS$ coherence message on the bus to inform all other cores and the shared cache about this request. Once the core receives the requested data, it moves to the S state. Similarly, if a core has a write request to a cache line in the I state, it issues a $GetM$ message on the bus and moves to the M state once data is received. The core is not required to take any action upon observing messages of other cores to a cache line that it has in the I state. Read requests to a cache line in the S state are hits and no message is broadcasted on the bus. In contrast, write requests to a cache line in the S state has to broadcast an Upg message on the bus to ask other cores to invalidate their local copies in their private caches since it is going to modify it. A core takes no action upon receiving an $OtherGetS$ from another core to a line that it has in the S state since multiple cores can simultaneously read the same cache line. Read and write requests to a cache line in the M state are hits and no message is broadcasted on the bus. If the core observes an $OtherGetS$ from another core requesting to read a cache line that it has in the M state, it sends the modified data to the requesting core and/or the shared memory and moves to the S state.

The Exclusive State. A commonly deployed optimization is to add the *Exclusive* (E) state to the coherence protocol constituting the MESI protocol. MESI is adopted by several COTS architectures targeting embedded systems such as the the NXP's T4080 [13] and the ARM's CHI. The semantic meaning of this state is as follows. If a cache line is in the E state, then it is valid, non-modified, and exclusive (i.e. not cached by another core). Two main advantages of the E state are as follows. 1) A core that has a store request for a cache line in the E state, does not need to wait for other cores to invalidate. Indeed, E carries the information that no other core has the cache 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. As a result, MESI protocol offers in general a better performance compared to MSI.

Since one of the main aspects of DISCO is that it provides predictability independent of the underlying protocol details, we apply DISCO for both the MSI and MESI protocols as examples.

5 Motivation

5.1 Performance Gains of Cache Coherence

Deploying cache coherence to orchestrate accesses to share data in real-time systems provides high performance gains compared to other approaches such as shared-data aware scheduling and private cache bypassing [7], [20]. In Figure 1, we show the execution time of both PMSI [7] and bypassing private caches entirely (or simply bypassing[1]). The applications used in this experiment are from the SPLASH3 benchmark suite [16], and the experimental setup is discussed in details in Section 8. As Figure 1 illustrates, PMSI outperforms bypassing for all benchmarks. Performance improvements reach up to $3.7\times$ (barnes) and $7\times$ (radiosity) with a geometric mean performance improvement across all benchmarks of $2\times$. This clearly represents promising results that motivate us to investigate cache coherence in the context of real-time systems.

5.2 Per-Request WCL

Despite its average-case performance gains, existing predictable cache coherence solutions suffer from large worst-case delays due to the introduced coherence interference [7], [10]. For instance,

1. Bypassing throughout this paper refers to skipping the access to the private cache and access directly the shared cache.

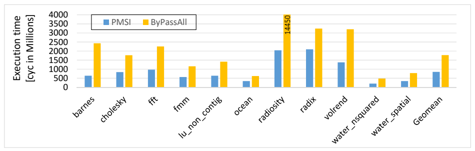


Fig. 1: Execution time.

TABLE 2: Worst-case latency components of private cache bypassing and PMSI techniques.

Latency Component	Bypassing	PMSI
Arbitration Latency	$N \cdot LLC^{acc}$	$N \cdot LLC^{acc}$
Coherence Latency	0	$N \cdot (2 \cdot N + 1) \cdot LLC^{acc}$
Access Latency	LLC^{acc}	LLC^{acc}

with bypassing, all cores pay the price of a shared cache access delay once granted access to the bus by the arbiter, regardless of the access pattern of other cores. This results in an access latency of one TDM slot, which we denoted as LLC^{acc} in Section 3 with no coherence latency at all. In addition to this access latency, for a system with N cores and a fair TDM arbiter, a request can suffer an arbitration latency up to one full TDM period or $ArbL^{WC,TDM} = N \cdot LLC^{acc}$. Table 2 summarizes these worst-case values. This is notably lower than the worst-case scenario under PMSI, where all cores compete to simultaneously access the same shared cache line. As Table 2 illustrates, in addition to the access latency and arbitration latency that is the same as those of bypassing, a memory request under PMSI suffers from a significant worst-case coherence latency. The value of this latency directly follows from [7]. From Table 2, total WCL of both bypassing and PMSI can be calculated as follows.

$$WCL^{PMSI}_{perReq} = 2 \cdot N^2 \cdot LLC^{acc} + 2 \cdot N \cdot LLC^{acc} + LLC^{acc} \quad (1)$$

$$WCL^{ByPass}_{perReq} = N \cdot LLC^{acc} + LLC^{acc} \quad (2)$$

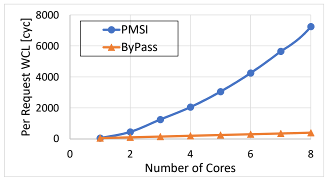
Figure 2a delineates this per-request WCL across different number of cores, which shows the significant gap between WCLs of cache coherent solution (PMSI) and bypassing solution due to coherence interference.

5.3 Time Compositionality

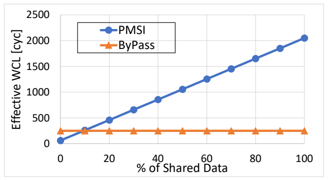
From Table 2 and Equation 1, we can rewrite the worst-case coherence as a function in arbitration latency as follows:

$$WCL^{PMSI,Coh}_{perReq} = (2 \cdot N + 1) \cdot ArbL^{WC,TDM} \quad (3)$$

This shows a coupling between the coherence and arbitration delays, where the coherence timing behavior is a function of the arbitration mechanism. This is not limited to PMSI but is also applicable to Other predictable protocols have similar dependency on the assumed arbitration such as CARP [10] and PISCOT [19]. It is also worth noting that if the TDM scheme is to be replaced by another arbiter, it is not necessarily that the only effect is to substitute the value of $ArbL^{WC,TDM}$ in Equation 3. The derivation itself, as indicated in these works is fundamentally dependent on the assumed arbitration policy and hence the relationship in Equation 3 will likely change as well. This clearly breaks timing compositionality of coherence since coherence timing behavior cannot be decomposed, studied and assessed in isolation of the arbitration, which is a requirement to achieve compositionality [1].



(a) Per-request WCL across different number of cores.



(b) Effective WCL for various percentage of shared data.

Fig. 2: Per-request WCLs (Equations 1 and 2) and effective WCLs (Equation 6).

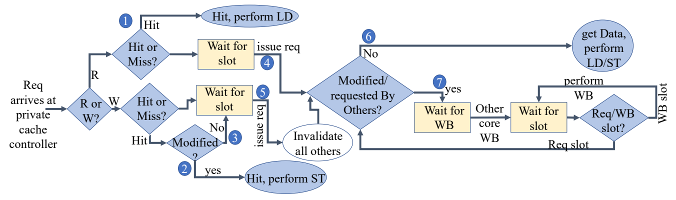


Fig. 3: PMSI flow diagram.

5.4 Total task's WCL

To bound the task's total Worst-Case Execution Time (WCET), the cumulative WCL over all requests generated by the task under analysis has to be computed. Towards this target, 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, $M(t)$ or simply M [2].

For bypassing, it is straightforward since all requests are serviced from the shared memory, every request can suffer the same WCL that is indicated in Equation 2. Therefore, the total WCL for by passing is computed as:

$$WCL^{Bypass}_{tot} = M \cdot WCL^{Bypass}_{perReq} = M \cdot LLC^{acc} \cdot (N+1) \quad (4)$$

For PMSI, it is more involved since requests to private (non-shared) cache lines need to be differently handled compared to requests to shared cache lines as the former will not suffer from coherence interference. Considering a partitioned cache hierarchy, where private and shared data are located in separate set such that shared data will not cause any conflict interference to private data, it is safe to assume that the access pattern (private hits and misses) to private cache lines (those not shared with other cores) can be analyzed in isolation and remains the same when the core suffers interference from other $N-1$ cores. Additionally, with this partitioning, from the task's analysis in isolation (either statically or experimentally), one can compute the number of requests to private cache lines (let it be $M^{private}$), and the number of requests to shared cache lines (M^{shared}) by examining the addresses of memory requests. Moreover, accesses to private cache lines can be further classified into hits and misses to the private cache, which we denote as $M^{private}_{hits}$ and $M^{private}_{misses}$, respectively. Unlike $M^{private}$, it is not possible to statically determine the hits or misses to the shared cache lines since this depends on the access behavior of other cores during run time, which can also access these shared lines. Therefore, the WCL has to be assumed for all accesses to shared lines. Assume the access latency to the core's private cache is L^{acc}_{priv} and recall that the WCL to access a

2. For readability, we drop the usage of t from the remainder of the paper. For instance, we use W instead of $W(t)$ to refer to the number of total writes generated by a core during time t.

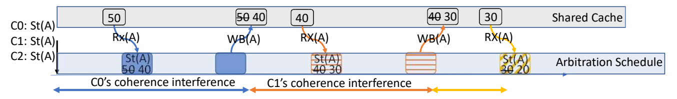


Fig. 4: Coherence interference in case of writes for PMSI. C2 is the core under analysis and it has to wait for both C0 and C1 before it gains access to block A.

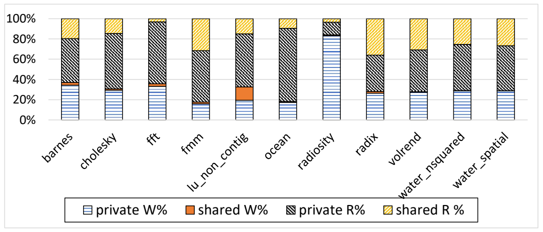


Fig. 5: Breakdown of Splash benchmark memory requests.

shared cache line (which includes coherence interference if exists) is WCL^{PMSI}_{perReq} as calculated by Equation 1. Accordingly, the cumulative total worst-case memory latency suffered by the task, WCL_{tot}, can be computed as:

$$WCL^{PMSI}_{tot} = M^{private}_{hits} \cdot L^{acc}_{priv} + M^{private}_{misses} \cdot (N+1) \cdot LLC^{acc} + M^{shared} \cdot WCL^{PMSI}_{perReq} \tag{5}$$

Dividing Equation 5 by the total number of task requests, we get the effective WCL of a single request (WCL_{eff}) as in Equation 6, which can be considered as the average WCL suffered by a single request to the cache.

$$WCL_{eff} = \%M^{private}_{hits} \cdot L^{acc}_{priv} + \%M^{private}_{misses} \cdot (N+1) \cdot LLC^{acc} + \%M^{shared} \cdot WCL^{PMSI}_{perReq} \tag{6}$$

To visualize this effect, Figure 2b plots the effective WCL for both bypass and PMSI for different percentage of accesses to shared data. Figure 2b shows that with increased percentage of shared data accesses, the gap between PMSI and bypass significantly increases. The reason for this behavior is that since the WCL^{shared}_{perReq} of PMSI (Equation 1) is much larger than that of bypass (Equation 2), increasing shared data accesses, this latency component will dominate the total WCL.

5.5 Distilling Coherence Effects on WCL

Now, our target is to reduce this high WCL resulting from cache interference. In doing so, we study carefully the effect of coherence across all access scenarios. We find that the high WCL is resulting from a pathological scenario and does not apply to all cases even for accesses to shared lines. This is a key observation and one of the main contributions of this paper; therefore, this subsection will discuss it in detail. We study all possible access scenarios in the existence of coherence, and plot these scenarios in Figure 3. Figure 3 follows the design guidelines of PMSI [7].

Hit Scenario. In case of a read hit (shown as event 1 in Figure 3) or a write hit to an already modified cache line in the private cache (at 2), the core proceeds with the load/store instruction without any arbitration or coherence delays. On the other hand, if the request is a write hit to a non-modified cache line (at 3), the core has to wait for a slot to access the shared bus as writes require to exchange coherence messages to invalidate copies of this line in all other private caches.

Miss Scenario. If the request is not available in the core's private cache, the core has to wait for its slot to issue this request on the shared bus. Once the core is granted a slot, it issues the request (4 in Figure 3 in case of a read, and 5 for a write). If the requested cache line is not modified in another core's private cache and no write requests are pending to this line, the core receives the data from the shared memory in the same slot and proceed to finish the load/store instruction. This is the scenario highlighted as 6 in Figure 3. On the other hand, if one of these two conditions is not satisfied, the core has to wait for all pending writes (if exist) to same line to finish first and the data to be updated in the shared memory (through a write back by another core) before it can obtain the requested line in its private cache. This is the scenario at 7 in Figure 3. As Figure 3 illustrates, the scenario at 7 is the one that triggers coherence interference and causes the largest delays.

Worst-Case Scenario. Based on this discussion, the pathological worst-case scenario is to assume that all cores in the system simultaneously ask to modify the same cache line. Accordingly, the request under analysis in the worst case has to wait for all other cores, where each core performs its access to obtain the cache line in its private cache, modifies it (performs the store instruction), and writes the new data back to the shared memory. This requires two memory transfers per each core of the other $N-1$ cores before the core under analysis is able to proceed with its access. Figure 4 depicts this scenario for a system with three cores, where the core under analysis is $C2$ and it has to wait for store requests from the other two cores before it can issue its own request. Under TDM scheduling, each transfer can wait for a complete TDM period (arbitration effects) before it can gain access to bus, where a TDM period is a function of N. This explains the quadratic effect of coherence interference on WCL.

Bypassing avoids this scenario by directly accessing the shared memory for every memory request, which eliminates the need for write backs, and hence, the coherence interference. However, this comes at the expense of not utilizing the private caches at all making every request suffering the large shared cache access time. This explains the performance degradation of bypassing compared to PMSI as discussed in 5.1. In contrast, we observe that the explained scenario can be avoided if only writes are made visible instantaneously to the shared memory, while reads do not cause any additional coherence interference. In Figure 4, if cores write directly to the shared memory, the resulting effect will be completely equivalent to bypassing independent of how reads are handled. The other important observation is that writes represent usually a small percentage of applications. Our analysis shows that across the SPLASH3 suite, writes represent on average 30% of the memory requests of the application as Figure 5 illustrates. The same observation holds for other benchmarks as well. For instance, we find that the PARSEC benchmark [26] suite and the EEMBC-auto [17] suite have on average 21% and 32% writes per application, respectively. We find the same observation proves true across the different SPEC benchmark suites (*int* and *fp*) and versions (2006 and 2017) as shown in [27].

6 Proposed Solution

Motivated by the observations we made in Section 5, we propose DISCO: a time-compositional discriminative coherence approach. The contributions of DISCO are three-fold. 1) It eliminates by design the worst-case scenarios covered in the previous section, and therefore, avoid its significant coherence delays. 2) It still maintains a high average-case performance by employing coherence and enabling simultaneous access to shared data. 3) It decouples the timing behavior of cache coherence and bus arbitration. Therefore, it enables time-composability in data-coherence real-time systems. This is achieved by providing time bounds of coherence interference that are independent of the details of the underlying bus arbitration policy. These three objectives are achieved by intentionally discriminating write memory requests by forcing them to update the shared memory immediately with any write (new) data from any core. This significantly simplifies the coherence protocol as it eliminates all the transient states needed because of data being updated privately by other cores such as in conventional protocols including MSI and MESI [11] or in predictable ones such as PMSI [7]. It is worth noting that MSI has 18 transient states and MESI has 20+ states. This originally created the uncertainties in the execution behavior, which either led to the unpredictable behavior of MSI and MESI protocols or with extremely pessimistic latency bounds for their predictable versions [20]. Figure 6 shows the simplifications that DISCO introduces to MSI and MESI protocols. Figure 6(a)/(c) is the original MSI/MESI protocol, while Figure 6(b)/(d) shows the effective protocol that MSI/MESI reduces to under DISCO. For readability, Figures 6(a) and 6(c) abstracts all the transient states between the stable states. It is worth noting that there are two different ways to realize DISCO, either by 1) realizing this simplified protocol in hardware, or 2) achieving the write bypass in already existing platforms through available support in these platforms. For now, we will detail the operation of DISCO, while we explain the required support in existing architectures that can allow for the realization of DISCO without redesigning the coherence protocol in Section 6.4. As Figures 6(b) and 6(d) illustrate, the M state is completely obsoleted since no core will have a cache line modified in its private cache without updating the shared cache. In addition, all transient states are also obsoleted. In other words, those states will never be visited under the operation of DISCO. This as shown as shaded states in the figures and corresponding transitions are dashed.

In addition to voiding certain states and transitions, DISCO effectively results in new transitions. These are shown in black in Figures 6(b) and (d) and are explained next. 1) If the request is a write to a line in the I state, it modifies this line directly into the shared cache and it does not allocate it to the private cache. Hence, it remains in the I state. Although allocating the cache line in the private cache might improve average performance by potentially allowing future read hits to this line, it requires an additional data transfer from the shared cache to the core, which increases the WCL. In particular, the slot width of the shared bus arbiter has to accommodate for two memory transfers instead of one. As a result, we choose not to allocate the line in this case to improve WCL. As explained in Section 4, a core with a cache line in the I state makes no change to its state as a response to events on this line by other cores. 2) If the core has a write request to a line that is in the S or E state, it has to issue a $GetM$ message on the bus to invalidate copies of this line in all other private caches and perform the write to its private cache as well as to the shared cache to keep it updated.

Leveraging these effective simplifications of the underlying coherence protocol, DISCO eliminates the large coherence delays due to write requests that modify data in private caches of cores while not being reflected on the shared memory. In other words, the long path in Figure 3 due to the modified/requested by others condition (the scenario at ⑦) is eliminated since this condition will be always false (no cache line will be modified in a core's private cache). Figure 7 illustrates the operation of DISCO. Since all writes are handled equally, we denote this approach as DISCO-AllW.

6.1 DISCO-AllW: Discriminative Coherence for All Cache Lines

A request to a cache line can be classified according to three factors.

1) **Request type.** With regard to the the instruction type, a request can be either a *read* (e.g. from a load instruction), or a *write* (e.g. from a store instruction).
2) **Data type.** This is related to the nature of the data stored in this cache line, it can be one of two possibilities: a *private* cache line (only accessed by the current task), or a *shared* cache line across tasks.
3) **Line state.** Finally, a request can be either a *hit* if the requested data exists (and is valid) in the private cache of the requesting core or a *miss* otherwise.

This results in a total of eight possibilities for any such request. DISCO-AllW operates on these cases based on four rules:

Rule 1. ***Operating Rules of*** DISCO-AllW.

(A) It does not distinguish between shared and private lines, both are treated equally.
(B) It treats all writes equally by sending them to the shared cache.
(C) Read hits are allowed and can proceed without requesting an access to the shared bus.
(D) Read misses have to wait for an access to the shared bus to obtain data from the shared cache.

Based on these rules, the aforementioned eight cases are reduced to only three scenarios under DISCO-AllW as illustrated in Figure 7a. Figure 8b depicts a flow chart for the operation of DISCO-AllW in all these three scenarios.

1) **Scenario ①: A Read Hit in the Private Cache.** Read hits are allowed immediately in the private caches and operate similar to traditional coherence protocols. This is because they do not require an access to the shared bus and do not result in any modification in the coherence state of the requested cache line.
2) **Scenario ②: A Read Miss in the Private Cache.** If the requested line is a read miss in the private cache, it has to be requested from the shared memory. Thus, the core has to wait for its slot and then issue its request on the shared bus. Since all writes are reflected immediately in the shared cache, the shared cache will always have the up-to-date data. Accordingly, the core will receive its requested line in the same slot and perform its load operation.
3) **Scenario ③: A Write Request.** As Figure 8b shows, any write request has to wait for an access slot to the shared bus to update the shared cache with the new data. In addition, all copies of the requested cache line in other cores' private caches have to be invalidated (since it is now outdated).

6.2 DISCO-SharedW: Discriminative Coherence for Shared Lines Only

DISCO-AllW operation does not make any assumption about the cache lines; in particular, it does not rely on the knowledge of

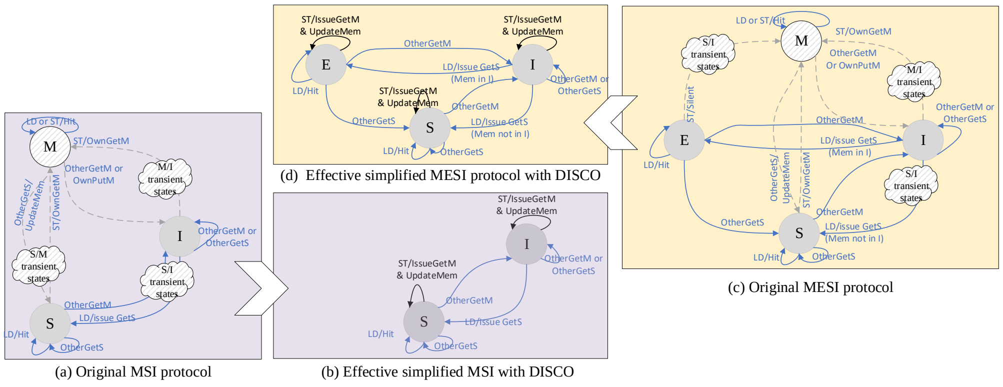
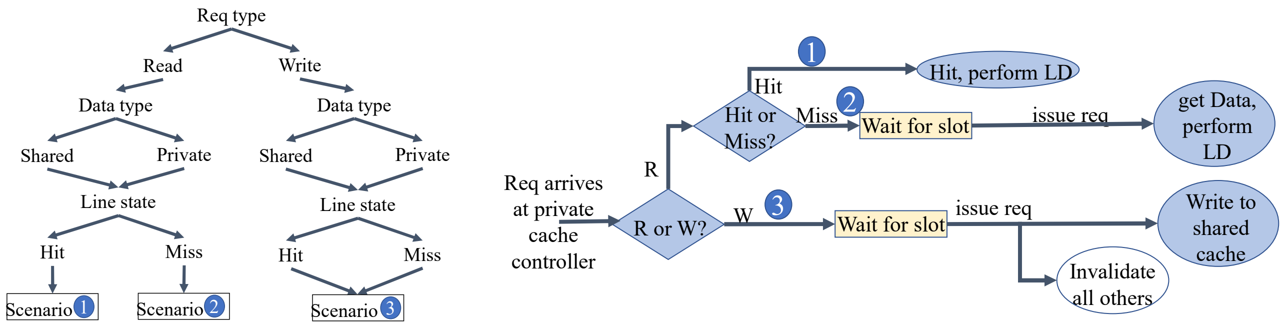
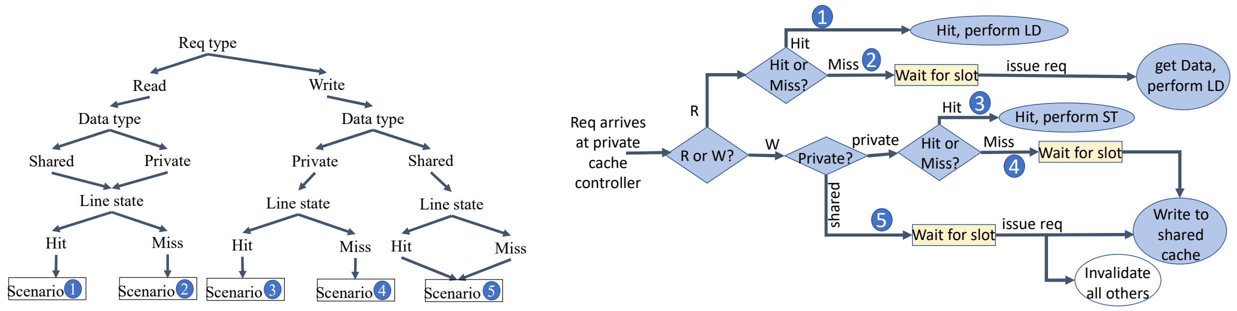

(a) Original MSI protocol (b) Effective simplified MSI with DISCO (c) Original MESI protocol (d) Effective simplified MESI protocol with DISCO

Fig. 6: Proposed DISCO coherence approach.



(a) Different access scenarios. (b) Flowchart of operations.

Fig. 7: Proposed DISCO-AllW coherence approach.



(a) Different access scenarios. (b) Flowchart of operations.

Fig. 8: Proposed DISCO-SharedW coherence approach.

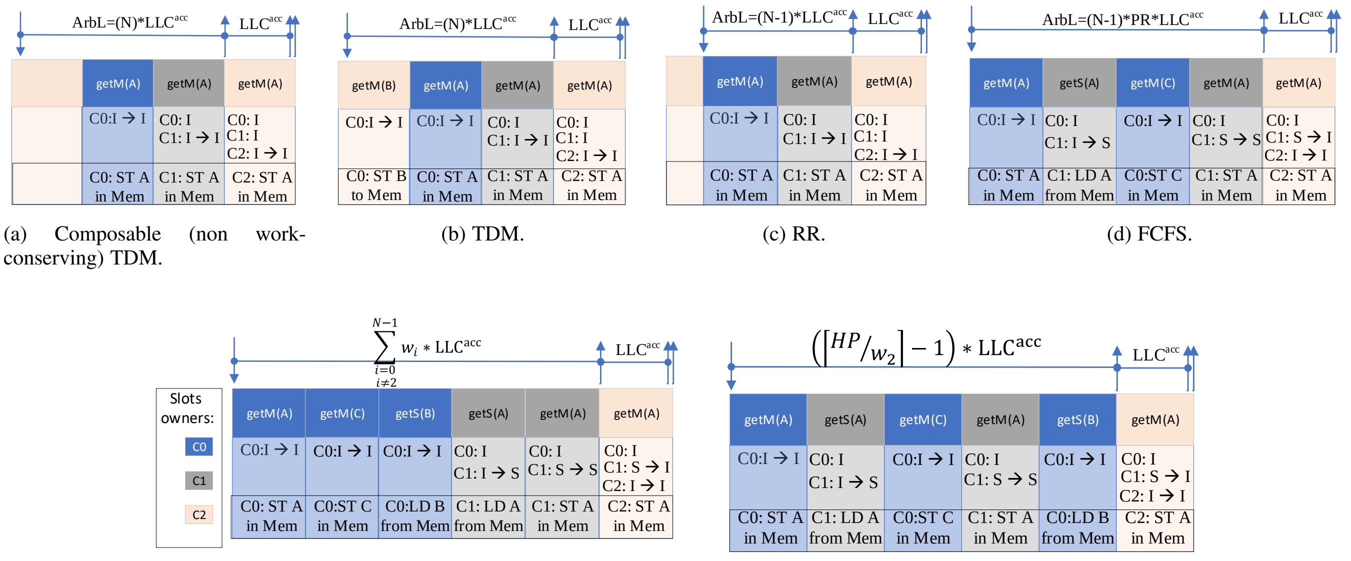
which lines are shared, which facilitates its adoption if such knowledge is not made available during execution. On the other hand, if such knowledge is available, we can improve the performance of the solution. This can be done by leveraging the fact that if a line is private for a task (and hence not shared among tasks), DISCO can safely allow write hits to this line without worrying about coherence interference. In doing so, we introduce another alternative to DISCO-AllW that we call DISCO-SharedW. Figure 9 illustrates the details of DISCO-SharedW, which operates according to the following rules.

Rule 2. ***Operating Rules of*** **DISCO-SharedW.**

(A) *Read hits are allowed to both private and shared lines and can proceed without requesting an access to the shared bus.*
(B) *Read misses have to wait for an access to the shared bus to obtain data from the shared cache.*
(C) *Write hits are allowed only to private lines that are not shared with other tasks. Those hits can proceed without requesting an access to the shared bus.*
(D) *Write misses to private lines has to wait for an access slot to the bus since it has to be requested from shared memory.*
(E) *Writes to shared lines have to go the shared cache.*

According to these rules, DISCO-SharedW handles reads exactly as in DISCO-AllW. On the other hand, writes are handled differently based on whether they are targeting a private or a shared cache line. This results in the following five scenarios of any memory request as depicted in Figure 8a.

1) **Scenario 1: A Read Hit in the Private Cache.**
2) **Scenario 2: A Read Miss in the Private Cache.** As illustrated in Figure 9 (compared to Figure 7), DISCO-SharedW handles Scenarios 1 and 2 exactly the same as DISCO-AllW.



(a) Composable (non work-conserving) TDM. (b) TDM. (c) RR. (d) FCFS.

(e) WRR. (f) HRR.

Fig. 9: Worst-case scenarios under different arbitration schemes. Core under analysis is $C2$.

3) **Scenario ❸: A write hit in the private cache for a private cache line.** Write hits to private lines are allowed based on Rule 2(C) and they execute immediately without the need to exchange any coherence messages.
4) **Scenario ❹: A write miss in the private cache for a private cache line.** Write misses to private cache lines are managed according to Rule 2(D) and they have to wait for an access slot to be sent to the shared memory.
5) **Scenario ❺: A write to a shared cache line.** Write hits to shared lines are still not allowed so as to avoid the high coherence delays resulting from it. Therefore, a write request to a shared cache line has to wait for an access slot to the shared bus since it has to update the shared cache (Rule 2(E)).

6.3 Bus Arbitration

Unlike state-of-the-art solutions offering predictable coherency [7], [9], [10], [19], [20], which are tightly coupled with a certain bus arbitration policy, DISCO is completely decoupled from the underlying arbitration scheme and can work seemingly with any predictable arbiter. To show this aspect, we deploy DISCO with various commonly used bus arbitration schemes and show both analytically and experimentally this independence. In particular, we focus the discussion on five arbitration techniques: TDM, RR, WRR, HRR, and FCFS. Figure 14 visualizes the operation of each of these arbiters in the worst-case scenario. Non work-conserving TDM in Figure 9a assigns a slot to each core, if the corresponding core does not have a request during this slot, the slot will remain idle. On the other hand, work-conserving TDM in Figure 9b will assign idle slots to the next core with a ready request. Both TDM versions have the same WCL discussed in Section 5 as also highlighted in the figures. RR in Figure 9c is more dynamic and it maintains a cyclic list for cores with ready requests resulting in a request suffering interference in worst case from $N-1$ other requests. For FCFS (Figure 9d) and assuming out-of-order cores with maximum number of possible pending requests of PR, a request has to wait in worst case for PR requests from $N-1$ other cores. Figure 9d exemplifies with $PR=2$, and hence a request from C2 has to wait for 4 requests from other cores in worst case. The WRR in Figure 9e assumes each core Ci is assigned a number of slots in the period equal to its weight w_i, and therefore, a request from a core j has to wait for $\sum_{i=0}^{N-1} w_i$ requests from other cores, where $i \neq j$. In the example in Figure 9e, w is 3, 2, and 1 for cores C0, C1, and C2, respectively. Hence a request from C2 has to wait for $3+2=5$ requests from cores C0 and C1, respectively in worst case. Finally, for HRR schedule (Figure 9f with a hyperperiod HP and w_i slots in the period for every core Ci, a request from core Ci might need to wait in worst case for $\lceil \frac{HP}{w_i} \rceil - 1$ requests before it can gain access. For example, in Figure 9f, a request from C2 waits for 5 slots before it can conduct the data transfer because $HP=6$, and w is 3, 2, and 1 for cores C0, C1, and C2, respectively. Equations 7–11 dictates the worst-case arbitration latency resulting from each of these arbiters.

$$ArbL^{WC,TDM} = N \cdot LLC^{acc} \tag{7}$$
$$ArbL^{WC,RR} = (N-1) \cdot LLC^{acc} \tag{8}$$
$$ArbL^{WC,FCFS} = (N-1) \cdot PR \cdot LLC^{acc} \tag{9}$$
$$ArbL_i^{WC,WRR} = \sum_{\substack{j=0 \\ j\neq i}}^{N-1} w_j \cdot LLC^{acc} \tag{10}$$
$$ArbL_i^{WC,HRR} = (\lceil \frac{HP}{w_i} \rceil) \cdot LLC^{acc} \tag{11}$$

6.4 Realization

A key advantage of DISCO is that it significantly simplifies the coherence protocol, while maintaining the average-case benefit of allowing tasks to simultaneously access coherent data. In this section we discuss how to realize DISCO. Since DISCO-AllW and DISCO-SharedW requires different supports, we separate the discussion on supporting each of the proposed two solutions as follows.

6.5 Realizing DISCO-AllW

There are different ways to enable DISCO-AllW.

6.5.1 *Pure hardware implementation:*

DISCO-AllW notably reduces the complexity of the coherence protocol as it eliminates all the transient states needed because of data being updated privately by other cores. Accordingly, this reduces the coherence protocol to the simple SI (or sometimes referred to as VI) protocol [11]. So, if a system is to be re-architected, one way to realize DISCO-AllW to achieve predictable, yet high-performance, coherence is to deploy the simple SI protocol.

6.5.2 *Utilizing support in existing architectures:*

Platforms supporting write-through caches natively support DISCO-AllW since all writes have to update the shared memory. Modern architectures allows users to configure memory regions with write-through caching. For instance, ARM allows the user to switch to write-through caches using a special register named Cache Behavior Override Register [28]. It is important to notice that this register controls the core's private cache only and is independent of the shared cache as implemented in the ARM1176JZ-S processor [28], which is again exactly the same behavior needed for DISCO. Intel processors also provides various control registers to support setting caches to different cache types including write-through [29]; nonetheless, it seems to apply the setting for all cache levels, which forces writes to be sent to the main memory.

6.6 Realizing DISCO-SharedW

Compared to DISCO-AllW, DISCO-SharedW requires additional support. This is because it requires to handle shared data (among cores) and private data (accessed by a single core) differently. Therefore, the trade-off here is providing better average performance at the expense of additional required support. DISCO-SharedW can be realized as a hardware/software codesign approach. Task/application developers need to have the ability to tag certain memory pages/regions as shared or private (or equivalently for DISCO-SharedW as write-back or write-through). We find that in many cases, shared data is an artifact of the application and can be identified statically. Examples of such cases include if such shared data is a concurrent/shared data structure in a parallel application [30]. Modern programming languages and operating systems also enable programmers to identify shared memory regions. An example is the `shmget()` and `mmap()` offered by Unix-based OSes and can be leveraged in C language. From embedded systems perspective, we foresee identifying shared data is also more amenable than in general-purpose systems due to the nature of the data flow in the system. Taking the automotive domain as an example, data coming from sensors and cameras are usually initially preprocessed by the corresponding processors associated to these sensors/cameras and then shared with other tasks that need to consume this data (to decide whether an action needs to be taken), which usually follows the model of a producer-consumer shared data pattern. In that case, the memory regions dedicated to this data (Whether in the form of a shared mailbox, queues, or buffers) can be tagged as shared.

The hardware, hence, need to have the support to configure, store, and then consume these tagged bits and perform the correct action: write-back for private data to increase performance, and write through for shared data to maintain tight latency bounds. Some modern architectures enable these extra tag bits per page as part of the Memory Managment Unit (MMU) or Translation Lookaside Buffer (TLB). The WIMG bits from NXP work as an example of such bits [31].

Furthermore, to be able to leverage the static cache analysis (hit vs miss rate) for private data, the latter has to be isolated from the shared data. Otherwise, shared data can impose additional misses to private data. Therefore, cache partitioning schemes are also required if improved task-level bounds are to be desired. More discussion about this is provided in Section 7.2.

Finally, if there are dynamic shared data that is not possible to know statically (due to non-trivial dynamic thread spawning or complex access patterns), in that case, DISCO would have to be conservative and assume all of this data is shared (to be safe from the analytical bound perspective).

7 Worst-Case Latency

In this section, we derive both the per-request as well as the total WCL for a system that deploys DISCO to manage shared data in its cache hierarchy.

7.1 Per-Request Worst-Case Latency

From the previous discussion, any memory request in either DISCO-AllW or DISCO-SharedW requires in the worst case only one memory transfer between the shared cache and the requesting core. For read requests, the shared cache sends data to the core, while for writes, the core sends updated data to the shared cache. Consequently, any memory request suffers only access and arbitration latencies. The excessive coherence delays discussed in Section 5 are completely eliminated.

Lemma 1. ***(Per-Request Worst-case Coherence Latency)****. A request from core Ci to a cache hierarchy deploying either versions of* DISCO *encounters a latency of at most LLC^{acc} from the time it is granted access to the bus until it receives its data regardless of the specifications of the deployed bus arbitration policy.*

$$WCL_{perReq}^{DISCO} = LLC^{acc} \tag{12}$$

Proof. The proof directly follows from the fact that under both versions of DISCO, the requested data for a request that misses in its private cache is always ready at the shared memory and by definition the access latency of the shared cache is LLC^{acc}. □

Lemma 2. ***(Total Per-Request Worst-Case Memory Latency)****. A request from core Ci to a cache hierarchy deploying either versions of* DISCO *encounters a latency that is at most:*

$$WCL_{perReq} = ArbL_i^{WC} + WCL_{perReq}^{DISCO} \tag{13}$$

7.2 Total Task's Worst-Case Memory Latency

Although both DISCO-AllW and DISCO-SharedW have the same per-request WCL, DISCO-SharedW improves the total WCL compared to DISCO-AllW. This is because leveraging the distinction between private and shared lines, a core's hit rate for private writes under interference from competing tasks is maintained the same as it is calculated in isolation. This is true since private lines by definition are not shared among tasks, and hence, do not experience interference from requests of tasks running on other cores. It is important to notice that although DISCO-AllW assumes that the knowledge of shared vs private lines is not made available to the hardware online upon execution, we can still use this information offline to derive the total WCL of the task.

Lemma 3. *Total worst case memory latency incurred by any task under* DISCO-AllW *can be calculated as:*

$$WCL_{tot}^{DISCO\text{-}AllW} = R_{hits}^{private} \cdot L_{acc}^{private} + (R_{misses}^{private} + R^{shared} + W) \cdot WCL_{perReq} \tag{14}$$

Proof. Based on the discussion of DISCO-AllW in Section 6.1, we prove Lemma 3 as follows.

Since all writes are treated equally, we denote write requests as simply W. By design, each one of these W requests has to wait for the corresponding core's slot to update the shared cache. Thus, they suffer the worst case scenario in Lemma 2 and each of them can have a WCL of WCL_{perReq}.

For the read requests to shared lines, denoted as R^{shared}: from Lemma 2, each one of those requests under DISCO-AllW suffers a WCL of WCL_{perReq}. Finally, since tasks do not interfere on private cache lines as aforementioned, tasks maintain the same hit rate calculated in isolation for read requests to these private lines. Accordingly, the number of read hits and misses to the private lines remain the same. Each one of the $R_{hits}^{private}$ encounters a hit latency of the private cache, $L_{acc}^{private}$, while every read miss has to access the shared cache encountering the scenario of Lemma 2 with a WCL of WCL_{perReq}. This constructs $WCL_{tot}^{DISCO\text{-}AllW}$ in Equation 14. □

Lemma 4. *Total worst case memory latency incurred by any task under* DISCO-SharedW *can be calculated as:*

$$WCL_{tot}^{DISCO\text{-}SharedW} = M_{hits}^{private} \cdot L_{acc}^{private} + (M_{misses}^{private} + M^{shared}) \cdot WCL_{perReq} \tag{15}$$

Proof. In DISCO-SharedW, requests (whether reads or writes) to private lines maintain their hit rate calculated in isolation. This entails any memory request to suffer one of three possible worst case scenarios as follows. Hits to private lines, $M_{hits}^{private} = R_{hits}^{private} + W_{hits}^{private}$, encounter the favorable private cache hit latency $L_{acc}^{private}$. Misses to private lines, $M_{misses}^{private}$, still has to wait for a slot to access the shared cache, and thus, suffers the WCL of WCL_{perReq} as per Lemma 2. Finally, Requests to shared lines, M^{shared}, also suffer the WCL of WCL_{perReq} since we cannot decide whether they are misses or hits as they are susceptible to interference from other tasks accessing same lines. Adding the WCL of these three scenarios lead to the $WCL_{tot}^{DISCO\text{-}SharedW}$ in Equation 15. □

7.3 On the Derivation of the Total WCL

7.3.1 Private and Shared Data

Equations 5, and 14–15, which derive the total WCL for PMSI, DISCO-AllW, and DISCO-SharedW, respectively, make an implicit assumption. They assume no conflict interference between shared and private data in the core's private cache (e.g. L1). As aforementioned, this can be achieved by partitioning the cache such that private and shared data are mapped to different memory spaces (e.g. different sets). Splitting memory address space to private and shared locations is an already existing approach to mitigate interference in the cache hierarchy [32]. However, if such partitioning is not possible, the analysis conducted in isolation to derive the miss and hit rates of a task's private data cannot be used. When running in a contending environment, private cache lines suffer additional conflict interference from shared data as they can be evicted because of the access pattern of shared cache lines that are mapped to the same cache line (under a direct mapped cache) or same set (under a set-associative cache). Therefore, to derive a safe bound, all private lines have to be declared misses. In this case, the total WCL will change to:

$$WCL_{tot}^{DISCO} = M \cdot WCL_{perReq} \tag{16}$$

Equation 16 can also be used when no information is available about the requests classification (i.e., misses or hits), and therefore, all requests have to be assumed misses. Finally, it is important to highlight that this only affects the calculated analytical total WCL, and it has no effect on the actual operation of different solutions during run time. In other words, it does not affect the average-case performance of DISCO. Per-request WCL also remains as previously calculated in Equation 13.

7.3.2 Reads and Writes

Another assumption that is made by Equation 14 is that it assumes the knowledge of the number of read and write requests made by the task. This information can be obtained from the task analysis (statically or dynamically) to obtain the number of load and store instructions [33]. Nonetheless, if such information is not available, Equation 16 can be used instead. Again, this does not affect the run-time behavior (and hence, average performance) of DISCO-AllW. It only affects the tightness of its derived bounds.

7.3.3 Effect of Write-backs in DISCO-SharedW

Since DISCO-SharedW allows write hits to private cache lines, those lines become dirty: they are modified in the core's private cache and are not updated in the shared cache. Hence, those lines need to be written to the shared memory at some point before they are evicted from the private cache due to replacement. The analysis in Section 7 for DISCO-SharedW does not take into account the effect of the write-back of these lines. Here, we discuss possible alternatives to account for this additional delay.

1) At the Request Level. In worst case, a miss request to the private cache initiates a replacement to a dirty cache line. This dirty line has to be written back to the shared memory before fetching the newly requested data. Moreover, this write back has to wait until the requesting core is granted access to the shared bus. As a result, a miss request encounters an additional arbitration latency due to this write back of the evicted line. This adds $ArbL^{WC}$ cycles to the WCL_{perReq} in Equation 13 in case of DISCO-SharedW. However, this delay is unnecessarily pessimistic since not every request is going to cause a replacement.

2) At the Task Level. Recall that the number of write-backs are because of writes in the private cache that are not updated at the shared memory. This number is obtainable for the task in isolation by using static analysis or experimental means since private cache is not shared among tasks running on other cores. We refer to the total number of write-backs initiated by a core during a period of time *t* as WB. For instance, a safe, but rather pessimistic, bound for WB is the total number of issued write requests to private cache lines during the same period t. This is true because write backs are initiated only because of dirty cache lines that are evicted, which in turn is bounded by the total number of writes to private lines. Shared lines cannot be dirty under DISCO-SharedW since similar to DISCO-AllW, they have to be sent directly to the shared memory. As a consequence, $WB = W^{private}$ is a safe bound. We say that this bound can be pessimistic, and hence, can be further tightened since a line can be written multiple times before it is evicted. However, obtaining an accurate value of the maximum number of WB is the concern of the analysis of the task in isolation, and is outside the scope of this paper. Lemma 5 calculates the new total WCL under DISCO-SharedW, while accounting for the delay effect of the write-backs due to cache replacement.

Lemma 5. *Total worst case memory latency incurred by any core Ci under* DISCO-SharedW *can be calculated as:*

$$\begin{aligned} WCL_{tot}^{DISCO\text{-}SharedW} &= M_{hits}^{private} \cdot L_{acc}^{private} \\ &+ (M_{misses}^{private} + M^{shared}) \cdot WCL_{perReq} \\ &+ ArbL_i^{WC} \cdot WB \end{aligned} \tag{17}$$

Proof. The proof directly follows from Lemma 4, while adding the last term to account for the write-backs effect. Since each one of those write-backs requests an access to the shared memory, it is subject to arbitration interference, which in worst case can result in a delay of $ArbL_i^{WC}$. This gives a total delay of $ArbL_i^{WC} \cdot WB$, which is the last term in Equation 17. □

It is worth noting that even with adding the write-back delays to the total WCL, DISCO-SharedW still provides a lower total WCL compared to DISCO-AllW. Comparing Equation 14 with 17, this is true for two reasons. 1) $WB \leq W^{private}$ as previously explained, and 2) $WCL_{PerReq} > ArbL^{WC}$.

8 Evaluation

To quantitatively evaluate the behavior of DISCO and compare it with state-of-the-art solutions, we simulate the behavior of a multi-core system with in-order pipelines, 8KB direct-mapped L1 per-core private cache, and a $1MB$ L2 shared cache across all cores. Cores are connected to the shared cache using a shared bus. Accesses to this shared bus are managed using a predictable arbiter. For a fair comparison with existing works, we use a TDM arbiter similar to them. In section 8.5, we study the effect of integrating both DISCO solutions with the different arbiters discussed in Section 6.3. The access latency of the L1 cache is 2 cycles, while access latency of the $L2$ is 50 cycles. To eliminate the large delays of off-chip memory access, similar to existing solutions [7], [10], we set $L2$ to be a perfect cache, i.e. all requests to $L2$ are hits. As discussed in Section 3, this is necessary to eliminate the effects of off-chip memory interference on the total execution time, which are orthogonal to this work. This is key to enable us to evaluate and scrutinize the analytically driven bounds in our experiments. Although a prefect LLC will affect average-performance results, we make the same assumption for all compared solutions. So, the average performance results should still be indicative of the trade-offs of different solutions. We deploy both versions of DISCO: DISCO-AllW and DISCO-SharedW. In addition, we also implement PMSI [7] and ByPassAll solutions discussed in Section 5. We use benchmarks from the SPLASH3 multi-threaded benchmark suite [17] as well as the EEMBC-Auto suite [17]. The simulation environment integrates with the Intel PIN tools [34] as follows. We run each benchmark through the PIN tool and collect execution traces that we run through the environment. For the SPLASH3 benchmarks, we run them using four threads in four cores (a thread for each core). For the EEMBC benchmarks, we use them to emulate a synthetic scenario that stresses the coherence effect. This is done by executing each of the EEMBC-auto benchmarks through the PINtool and feed the collected trace to each of the four cores in the environment. Doing so, all data is shared across all cores, which signifies the coherence interference.

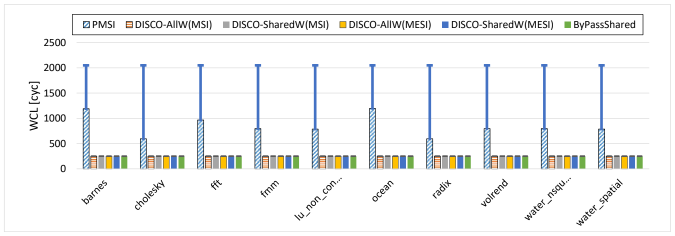
8.1 Per-Request Worst-Case Latency

Figure 10 delineates the WCL for any request to the cache hierarchy in a four-core system for both SPLASH3 (Figure 10a) and EEMBC (Figure 10b). The figure shows both the analytical WCL bounds (T bars) and the the observed (experimental) WCL (colored solid bars) for both PMSI and the two versions of DISCO. From this experiment, we make the following observations. 1) DISCO reduces the analytical WCL by $7.2\times$ compared to PMSI. The analytical WCL of PMSI is 2050 cycles compared to 250 cycles in DISCO. 2) PMSI incurs a large gap between experimental and analytical WCLs. In the SPLASH3 benchmarks (Figure 10a), this gap ranges from 70% (barnes and ocean) and reaches up to $3.4\times$ (*cholesky* and *radix*). This is because PMSI's analytical WCL assumes a pathological worst-case scenario that is hard to construct in real applications as explained in Section 5. Even with the synthetic experiments of EEMBC (Figure 10a), the gap is more than 45% for most benchmarks. On the other hand, DISCO's analytical and experimental WCLs are identical indicating the bound tightness. DISCO achieves this tightness by deliberately avoiding the large-latency scenarios created by write requests in private caches without updating the shared memory. 3) It is worth noting that DISCO achieves the same WCL as BypassAll solution (not shown in Figure 10), while still allowing read hits to the private caches, which improves both total WCL and average performance as we discuss in the next subsections.

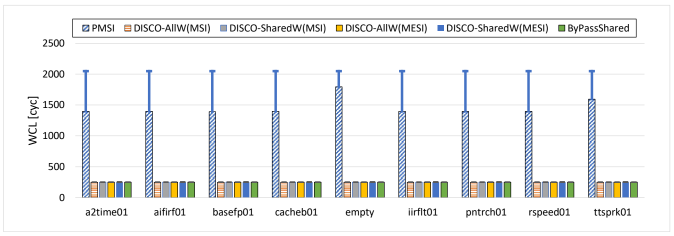
8.2 Total WCL

Figure 11 delineates the total WCL of all the evaluated approaches for the SPLASH-3 benchmarks. To facilitate readability, all results in Figure 11 are normalized to the total WCL of the ByPassAll approach. Recall that the total WCL is the worst-case memory latency that is suffered by a core during a time period *t* and is calculated in Equations 4, 5, 14, and 17 for ByPassAll, PMSI, DISCO-AllW, and DISCO-SharedW, respectively. Figure 11 introduces several interesting observations.

1) PMSI encounters the largest total WCL. This is due to the quadratic effect of coherence interference as we discussed in details in Section 5. The normalized PMSI's total WCL varies per benchmark based on the percentage of shared data. For instance, the *radix* benchmark suffers the maximum value of PMSI's total WCL ($3.3\times$ ByPassAll's). Investigating the reason for this, we found that *radix* has the maximum percentage of shared data (around 38%). Accordingly, from Equation 5, the term that suffers the maximum latency of WCL_{perReq}^{PMSI} dominates the total WCL. Interestingly, there are cases where PMSI has a lower total WCL than ByPassAll. Namely, this is the case for the *fft* and *radiosity* benchmarks in Figure 11. Analyzing both benchmarks, we found that both benchmarks in contrast to the *radix* benchmark have the maximum percentage of private (non-shared) data: 94% and 96% for *fft* and *radiosity*, respectively. This enables PMSI to leverage hits to this non-shared data, which gives it an advantage over ByPassAll, which forces all requests to go to the shared memory. 2) Compared to PMSI, DISCO-SharedW achieves up to 6x tighter total WCL (*barnes*) and 3.5x on average. DISCO-AllW, on the other hand, has up to 3.3x tighter total WCL (*fmm*) and 1.95x on average. PMSI has a lower total WCL than DISCO-AllW in



(a) Splash3.



(b) EEMBC.

Fig. 10: Both analytical (T bars) and experimental (solid bars) per-request WCL.

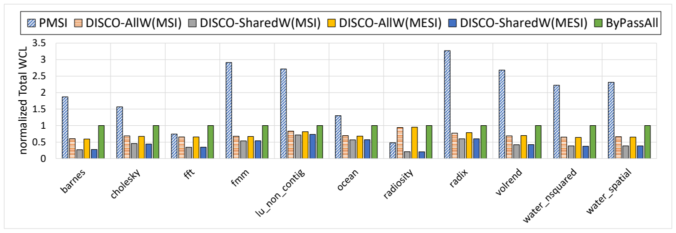


Fig. 11: Total worst-case latency of Splash3.

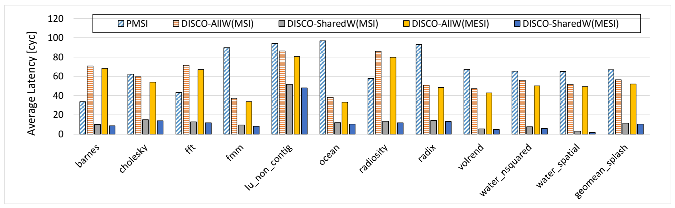


Fig. 13: Average latency of Splash3.

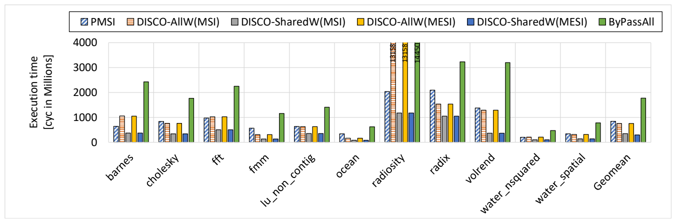


Fig. 12: Execution time.

case of *fft* and *radiosity* benchmarks for the same reasons as in observation 1 because DISCO-AllW does not allow write hits. An extended discussion about the behavior of these two benchmarks is provided in Section 8.4. 3) Although DISCO-AllW and DISCO-SharedW offer the same per-request WCL of ByPassAll, both proposed approaches provide a tighter total WCL than ByPassAll. The reason for that is that both solutions allow read hits in cores' L1 caches, while DISCO-SharedW also allows write hits to core's private (non-shared) cache lines. This improves the total WCL since as proved in Lemmas 3-5, those hits will not suffer the arbitration latency due to contention on the shared bus. This enables DISCO-AllW to provide up to 65% (*barnes* benchmark) and 42% on average tighter total WCL than ByPassAll. Furthermore, DISCO-SharedW provides up to $3.8\times$ (*radiosity*) and $1.5\times$ on average tighter WCL compared to ByPassAll.

8.3 Average Performance: Execution Time

Figure 12 depicts the overall execution time for SPLASH3 benchmarks under four different approaches: PMSI, ByPassAll (all requests access L2), and both versions of DISCO. From this experiment, we make the following observations. 1) Compared to ByPassAll, DISCO-AllW improves performance (reduced execution time) by up to $2.8\times$ and $1.5\times$ on average. Recall from Section 8.1 that DISCO achieves same WCL as ByPassAll, this verifies the ability of DISCO to balance WCL and performance. 2) DISCO-AllW also has a better overall performance compared to PMSI (up to 100% and 12% better performance on average). Nonetheless, PMSI has better performance than DISCO-AllW for four benchmarks: *barnes, fft, radiosity*, and *water_nsquared*. We discuss the reasons behind these results in more details later in Section 8.4. 4) Figure 12 clearly illustrates the benefits of DISCO-SharedW. DISCO-SharedW outperforms all other approaches for all benchmarks: it achieves up to $3.2\times$ and $1.6\times$ on average better performance than PMSI, more than $11\times$ and $5\times$ on average better performance than ByPassAll, and $2\times$ on average better performance than DISCO-AllW. Again, using either version of DISCO depends on the system capabilities. If the system has the capabilities (either in software or hardware) that isolates between shared and unshared data, DISCO-SharedW represents a promising design choice. Contrarily, if the system is not able to distinguish shared data, DISCO-AllW is the best available choice.

8.4 Average Performance: Average- Memory Latency

To further study the average-case performance behavior of DISCO compared to PMSI, we show the average-case memory latency for SPLASH3 benchmarks in Figure 13. Figure 13 confirms the same behavior observed in the execution time in Figure 12. 1) DISCO-AllW outperforms PMSI on average by 18%, while PMSI achieves better performance for some benchmarks; *namely, barnes*, and *fft*. 2) DISCO-SharedW, on the other hand, considerably outperforms PMSI and achieves up to $12\times$ and $5.8\times$ on average less average latency. The intuition behind such behavior of benchmarks where PMSI outperforms DISCO-AllW is that they exhibit larger number of write hits to private cache lines compared to other benchmarks. Because DISCO-AllW forces all writes to access the shared cache, it does not leverage this temporal locality characteristic of such benchmarks; hence, it incurs worse overall performance. In contrast, DISCO-SharedW does leverage this locality by allowing write hits to private cache lines and hence, achieves better performance. To investigate this theory, we deploy performance counters in the simulation environment to count the number of write hits both to private and shared cache lines under PMSI. Data collected from these counters confirm our explanation that PMSI achieves better performance for those benchmarks that exhibit high number of write hits to private lines. Nonetheless, for those benchmarks, DISCO-SharedW still achieves better performance than PMSI.

8.5 DISCO with Different Arbitration Policies

As aforestated, one of the main advantages of DISCO over state-of-the-art coherent solutions [7], [9], [10], [19], [20] is that it is time-compositional and decouples the effects of coherence from the specifics of the deployed arbitration policy. To investigate this advantage, we deploy both versions of DISCO: DISCO-AllW and DISCO-SharedW on systems implementing different arbitration policies. Namely, we test the operation of DISCO using the following arbiters: TDM, RR, WRR, HRR, and FCFS. Figure 14 delineates both the measured as well as analytical worst-case latencies for each of these arbiters using a four-core system and the EEMBC setup discussed before. The core under analysis in this figure is C0. Since WRR, HRR, and FCFS can be configured using different service assignments. We pick the following assignment for each of them. For WRR, WRR-4444 indicates that each of the four cores gets four consecutive slots to access the bus. For HRR, HRR-2211 indicates that cores C0 and C1 are granted two slots each in the harmonic period, while cores C2 and C3 are granted one slot each. Finally, for FCFS, FCFS-4 indicates that each OOO cores is allowed to have a maximum of four-requests in-flight. We make the following observations from Figure 14. 1) All observed latencies are within their corresponding analytical bounds. This confirms that DISCO provides the promised bounds when combined with any of these arbiters. 2) For RR, TDM, and HRR-2211, the analytical latency was observed during execution, which means these arbiters offer tight latency bounds for core C0. This is because under each of these arbiters, C0 has to wait for only one request from every other core (plus one in case of TDM), which is a scenario that is likely to happen in practice. 3) For both WRR-4444 and FCFS-4, we observe a gap between observed and analytical latencies. This is because the analytical bound has to consider the worst-possible scenario: a request from Core C0 has to wait for 4 requests from each other core. Although this is a possible scenario, it is less likely to happen compared to the one discussed in observation 2. Since WRR, HRR, and FCFS depends on how service assignment is allocated among cores, we conduct another set of experiments exploring the effect of different assignments on the worst-case latency both experimentally and analytically. Results of these experiments are shown in Figure 15. Similar to the previous experiment, core under analysis is C0. For WRR and HRR (Figures 15(a) and (b)), the number in the legend refers to the number of slots assigned to each core, C0-C4, respectively, in the schedule. For example, HRR-3111 means that C0 gets three slots in the period, while every other core gets only a single slot. Please note that assigning one slot per core for all cores, reduces WRR (WRR-1111) and HRR (HR-1111) to normal RR. Therefore, in Figure 15(a) for WRR, we experiment with assigning more than one slot per core: WRR-2222 and WRR-4444. We also experiment with uneven slot distribution. In particular, we configure WRR to give two slots to C0 (core under analysis) vs. one slot to every other core. From the results in Figure 15(a), we observe that WRR does not indeed improve the WCL of the C0 regardless of the slot distribution. Intuitively, this is because as far as every other core is granted at least one slot in the schedule to access the bus, the best WCL that can be achieved is equivalent to this guaranteed by RR: waiting for one request from every other core in worst

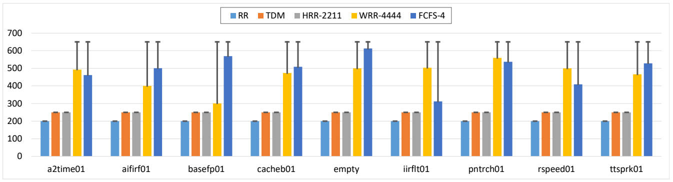


Fig. 14: Measured (solid colored bars) and analytical (T-sharp bars) WCL for different arbiters (y-access is in cycles) .

case. In contrast, HRR (Figure 15(b)) is capable of improving the WCL of the core under analysis thanks to its distribution of the slot assignment as explained in Section 6.3. For instance, HRR-3111 in Figure 15(b) has a WCL of 100 cycles for C0's requests. This is because in this schedule C0 is granted access to the bus every other slot, i.e. a request from C0 has to wait in worst case for only a single request (coming from another core). It is important to notice that the trade-off here is that such schedule loses fairness since it penalizes other cores. Such unfair schedulers are commonly used in mixed-criticality systems, where cores have different criticalities [35]. Finally, for FCFS, we sweep the maximum number of allowed pending requests from each core as 1, 2, or 4. AS Figure 15(c) shows, increasing the number of possible pending requests increases the unpredictability in the system by 1) increasing the analytical bound, and 2) increasing the gap between the maximum observed latencies in practice compared to the analytical bound.

8.6 Effect of Non-Perfect LLC

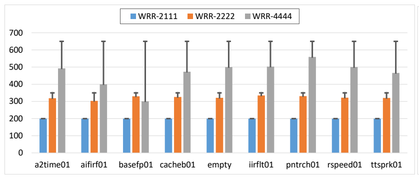
In this experiment, we study the effect of a non-perfect LLC on the different predictable solutions. In doing so, we experiment with the same setup as before, except with the following changes: 1) LLC is a non perfect 8-way set associative cache and deploys a least recently used (LRU) cache replacement policy, and 2) misses from the LLC goes to a main memory. We model this main memory as a fixed latency with a 200 cycles access time. Figure 16 shows the results of the different solutions for this experiment. Results show that Similar trend as in Figure 12, where ByPassAll has the worst and DISCO-SharedW has the best execution time.

9 Conclusion

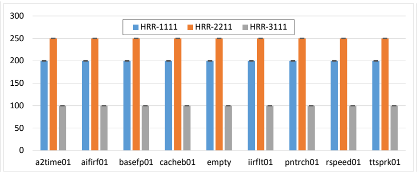
We propose DISCO: a discriminative coherence protocol that significantly reduces the coherence delays, and hence, provides tighter bounds than existing predictable coherence protocols. DISCO also achieves a high average-case performance by allowing tasks to simultaneously cache and access data in the cores' private caches. This is achieved by eliminating the scenarios that cause high coherence interference under traditional coherence protocols. We discuss the realization of DISCO using features offered by commodity systems. DISCO achieves up to $7.2\times$ tighter latency bounds than existing predictable coherence protocols, while improving performance by up to $11.4\times$ ($5.3\times$ on average) compared to competitive cache bypassing techniques.

References

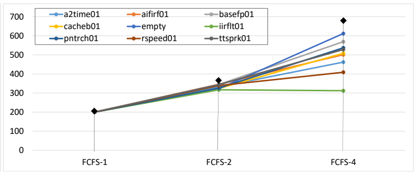
[1] S. Hahn, J. Reineke, and R. Wilhelm, "Towards compositionality in execution time analysis: definition and challenges," *ACM SIGBED Review*, vol. 12, no. 1, pp. 28–36, 2015.



(a) WRR.



(b) HRR.



(c) FCFS.

Fig. 15: Measured (solid colored bars in (a) and (b), and solid lines in (c)) and analytical (T-sharp bars in (a) and (b), and diamond black dots in (c)) worst-case latencies (y-access is in cycles) with different arbiter parameters .

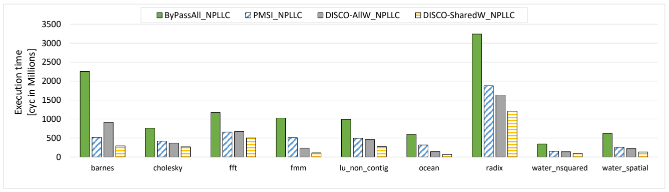


Fig. 16: Execution time under non-perfect LLC.

[2] K. Yang and Z. Dong, "Mixed-criticality scheduling in compositional real-time systems with multiple budget estimates," in *Proceedings of the 41st IEEE Real-Time Systems Symposium (RTSS)*, 2020.
[3] C. H. Fleming and N. G. Leveson, "Improving hazard analysis and certification of integrated modular avionics," *Journal of Aerospace Information Systems*, vol. 11, no. 6, pp. 397–411, 2014.
[4] M. Chisholm, N. Kim, B. C. Ward, N. Otterness, J. H. Anderson, and F. D. Smith, "Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2016.
[5] A. Bansal, J. Singh, Y. Hao, J.-Y. Wen, R. Mancuso, and M. Caccamo, "Cache where you want! reconciling predictability and coherent caching," *arXiv preprint arXiv:1909.05349*, 2019.
[6] G. Gracioli and A. A. Fröhlich, "On the design and evaluation of a real-time operating system for cache-coherent multicore architectures," *ACM SIGOPS Oper. Syst. Rev.*, 2015.
[7] M. Hassan, A. M. Kaushik, and H. Patel, "Predictable cache coherence for multi-core real-time systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.
[8] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte, "Contention-free execution of automotive applications on a clustered many-core platform," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.
[9] N. Sritharan, A. M. Kaushik, M. Hassan, and H. Patel, "Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems," 2019.
[10] A. M. Kaushik, P. Tegegn, Z. Wu, and H. Patel, "Carp: A data communication mechanism for multi-core mixed-criticality systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2019.
[11] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, 2011.
[12] M. Hassan, "Discriminative coherence: Balancing performance and latency bounds in data-sharing multi-core real-time systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, Jul. 2020, pp. 1–22.
[13] Freescale semicondutor, "QorIQ T2080 Reference Manual," 2016, Also supports T2081. Document Number: T2080RM. Rev. 3, 11/2016.
[14] D. Radack et al. (Rockwell Collins), "Civil Certification of Multi-core Processing Systems in Commercial Avionics," 2018.
[15] B. Lesage, D. Hardy, and I. Puaut, "Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures." in *International Conference on Real-Time and Network Systems*, 2010.
[16] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2016, pp. 101–111.
[17] J. Poovey *et al.*, "Characterization of the EEMBC benchmark suite," *North Carolina State University*, 2007.
[18] D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches," in *IEEE Real-Time Systems Symposium (RTSS)*, 2009.
[19] S. Hessien and M. Hassan, "The Best of All Worlds: Improving Predictability at the Performance of Conventional Coherence with No Protocol Modifications," in *IEEE Real-Time Systems Symposium (RTSS)*, Oct. 2020, pp. 1–12.
[20] A. M. Kaushik, M. Hassan, and H. Patel, "Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems," *IEEE Transactons on Computers (TC)*, pp. 1–23, Oct. 2020.
[21] N. Sensfelder, J. Brunel, and C. Pagetti, "Modeling cache coherence to expose interference," in *ECRTS 2019*, 2019.
[22] ——, "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, 2020.
[23] M. M. MARTIN, M. D. HILL, and D. J. SORIN, "Why on-chip cache coherence is here to stay," *Communications of ACM*, 2012.
[24] D. Guo, M. Hassan, R. Pellizzoni, and H. Patel, "A comparative study of predictable dram controllers," *ACM Transactions on Embedded Computing Systems (TECS)*, 2018.
[25] S. Hahn, M. Jacobs, and J. Reineke, "Enabling compositionality for multicore timing analysis," in *Proceedings of the 24th international conference on real-time networks and systems*, 2016, pp. 299–308.
[26] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.
[27] A. Limaye and T. Adegbija, "A workload characterization of the spec cpu2017 benchmark suite," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2018, pp. 149–158.
[28] ARM, "ARM arm1176jz-s technical reference manual," 2013.
[29] Intel, "Intel 64 and IA-32 architectures software developer's manual," *Volume 3A: System Programming Guide, Part*, vol. 1, no. 64, 64.
[30] M. Moir and N. Shavit, "Concurrent data structures." *Handbook of Data Structures and Applications*, vol. 47, pp. 1–47, 2004.
[31] NXP, "Memory Management Unit," https://www.nxp.com/docs/en/supporting-information/TPQ2CH09.pdf, 2021.
[32] B. Lesage, I. Puaut, and A. Seznec, "PRETI: Partitioned real-time shared cache for mixed-criticality real-time systems," in *Proceedings of the 20th International Conference on Real-Time and Network Systems (RTNS)*, 2012.
[33] S.-K. Kim, S. L. Min, and R. Ha, "Efficient worst case timing analysis of data caching," in *Proceedings Real-Time Technology and Applications*. IEEE, 1996, pp. 230–240.
[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.
[35] M. Hassan and H. Patel, "Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

Mohamed Hassan is an Assistant Professor at McMaster University, Hamilton, ON, Canada. He received the M.Sc. degree from Cairo University, Egypt; and the Ph.D.degree from the University of Waterloo, Waterloo, Canada. His current research interests include real-time embedded systems, computer architecture, memory systems, hardware validation, and security.