Discriminative Coherence: Balancing Performance and Latency Bounds in Data-sharing Multi-Core Real-Time Systems

Mohamed Hassan
McMaster University
mohamed.hassan@mcmaster.ca

Abstract

Tasks in modern multi-core real-time systems share data and communicate among each other. Nonetheless, the majority of published 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, approaches that bypass private caches to avoid coherence interference altogether suffer from significant average-case performance degradation. On the other hand, proposed predictable cache coherence protocols increase the worst-case memory latency (WCL) quadratically due to coherence interference. In this paper, by carefully analyzing the scenarios that lead to high coherence interference, we make the following observation. 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 balance average-case performance and WCL. This is achieved by disallowing modified data in private caches, and hence, the significant coherence delays resulting from them are avoided. In addition, DISCO achieves high average performance by allowing tasks to simultaneously read shared data in the private caches. Moreover, if the system supports the distinction between private and shared data, DISCO further improves average performance by allowing for the caching of private data in cores' private caches regardless of whether it is modified or not. Our evaluation shows that DISCO achieves $7.2\times$ lower latency bounds compared to the state-of-the-art predictable coherence protocol. DISCO also achieves up to $11.4\times$ ($5.3\times$ on average) better performance than private cache bypassing for the SPLASH-3 benchmarks.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Real-time systems; Computer systems organization → Architectures

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

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.15

1 Introduction

Demands from modern applications of embedded systems such as those in automotive, avionics, industrial automation and healthcare are shaping the research directions in real-time embedded systems. The high performance demands from these applications ignited the transition from single-core to multi-core real-time systems [37]. In addition, meeting the high data demand was a strong motive behind exploring the adoption of complex memory hierarchies composed of multiple levels of caches [44] and include shared caches [10, 13, 25, 36, 29], shared interconnects [7, 15, 19] as well as off-chip memories [9, 12, 18, 22] instead of the small-sized static on-chip memories found in traditional low-end embedded systems. Despite this large volume of research, one demand from the aforementioned applications is yet to be efficiently addressed: allowing a 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 [6]. This is mainly because data sharing is problematic and can lead to significant interference delays [3, 11] or even unpredictable behaviors [14] if it is not carefully addressed.

32nd Euromicro Conference on Real-Time Systems (ECRTS 2020).
Editor: Marcus Völp; Article No. 15; pp. 15:1–15:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

On the other hand, researchers have already realized the importance of enabling data sharing in the context of real-time systems [4, 6, 11, 14, 16]. 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 [6], data-aware scheduling [4, 11], or bypassing caches [6, 3, 26]. 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, [14, 39, 40, 23] propose predictable cache coherence protocols. The problem with these protocols is that they require complex changes to cache controllers and more importantly, they result in a significant increase in the worst-case latency (WCL) upon accessing memory due to coherence interference. In this paper, we propose DISCO: a discriminative coherence solution that addresses the aforementioned drawbacks. Towards this target, we make the following contributions.

1. We exhaustively study 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.

2. Based on these two observations, we propose DISCO to prohibit the caching of modified data in the cores' private caches. All data modifications are carried out at the shared cache. DISCO intentionally discriminates against write memory requests since they are forced to access the shared cache even if data already exists in the requesting core's private cache, while 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.

3. To further improve the system performance, we also propose 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 7.3.3.

4. We conduct a detailed analysis to calculate the WCL incurred by any memory request as well as the total WCL for both DISCO-AllW and DISCO-SharedW (Section 7).

5. We compare both versions of DISCO with the two state-of-the art competitive solutions: PMSI coherence protocol [14] and cache bypassing [26, 3]. Our evaluation uses both the SPLASH-3 [35] parallel data-sharing benchmarks as well as synthetic experiments that are based on the EEMBC-Auto benchmarks [33]. 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.

	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×

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

2 Related Work

With the adoption of multi-core architectures in real-time embedded systems being on the rise, several new challenges face the researchers in these systems. Predictably managing the shared hardware components among different cores (such as interconnects, on-chip caches, and off-chip memories) is one of the biggest challenges. This is because processing elements in multi-core architectures compete to access these resources which results in significant interference in the system. To address this challenge, several recent research efforts aim at providing predictable access to shared interconnect [44, 7, 15, 31, 19], shared caches [42, 36, 43, 10], and shared DRAM [32, 34, 1, 9, 22, 17, 12]. While these efforts successfully address the timing interference problem, the data interference problem is usually overlooked.

Most of the aforementioned solutions adopt the independent-task model, where tasks do not share data. Recently, researchers recognized the importance of data sharing and proposed solutions to handle it [6, 11, 14, 3, 4, 26, 39, 23, 40]. 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 [4]; hence, they do not compete for shared data; 1.2) assigning tasks with shared data to the same core [6]; 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 [11]. This direction enforces new constraints on the system scheduler interference, which deteriorates system schedulability [40]. 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 [13, 26] but is then used to avoid coherence interference of shared data [6, 3]. 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 using cache coherence protocols [14, 39, 23, 40]. Cache coherence is notoriously the main solution adopted by commercial-of-the-shelf (COTS) multi-core architectures [30, 41]. 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 [14, 40]. On the other hand, it suffers a notably high worst-case memory latency due to the introduced coherence interference. For instance, PMSI [14] has 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.

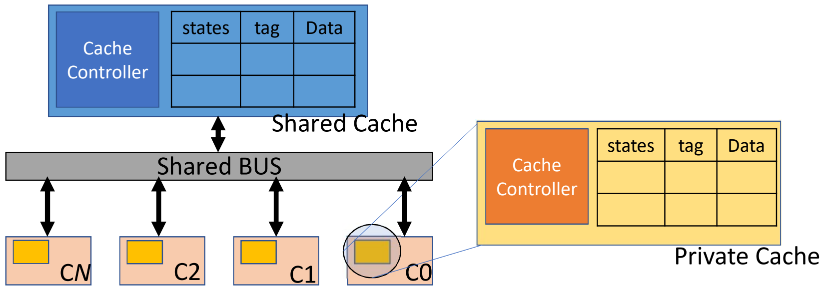


Figure 1 System model.

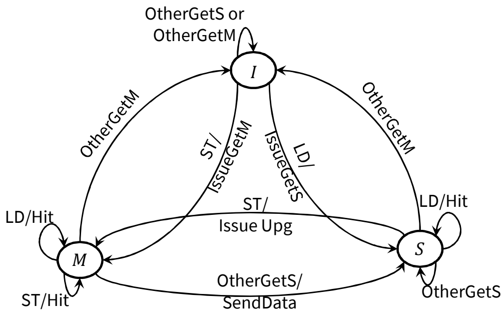


Figure 2 MSI coherence states, messages, and transitions. Messages observed by the core on the bus from other cores are indicated as $Other$ (e.g. $OtherGetS$).

3 System Model

We assume the multi-core architecture depicted in Figure 1, 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). LLC is accessed through a shared bus. Cores can also share an off-chip memory. We also assume that timing interference is resolved in the shared cache using partitioning or coloring [10], and in shared main memories using existing solutions orthogonal to this work [12, 8].

Bus Arbitration. Without loss of generalization, we assume that accesses to the shared bus are managed according to a Time Division Multiplexing (TDM) scheme. Solutions proposed in this paper are independent of the deployed arbiter and can be applied to other arbiters as well. However, the timing analysis we perform in Section 6 assumes a TDM bus arbitration. Similar to existing work [14, 39], we set the slot width to accommodate for the one data transfer between private and shared caches in addition to coherence messages. This slot width is denoted as L_{acc}^{miss} since it is incurred if a request misses in the corresponding core's private cache.

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. The Modified-Shared-Invalid (MSI) is considered the baseline coherence protocol [38], 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 protocl deployed in Intel's i7 and the MOESI protocol deployed in AMD's Opteron [20]. Therefore, we use it as a mean to explain the basics of a coherence protocol.

Figure 2 depicts the three states of MSI as well as all possible transitions between them. 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 $OtherGetS$ on the bus 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.

5 Motivation

5.1 Performance Gains of Cache Coherence

PMSI [14] provides high performance gains compared to other approaches such as shared-data aware scheduling and private cache bypassing by deploying cache coherence to orchestrate accesses to share data. In Figure 3, we show the execution time of both PMSI and bypassing private caches entirely (or simply bypassing[1]). The applications used in this experiment are from the SPLASH3 benchmark suite [35], and the experimental setup is discussed in details in Section 8. As Figure 3 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.

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

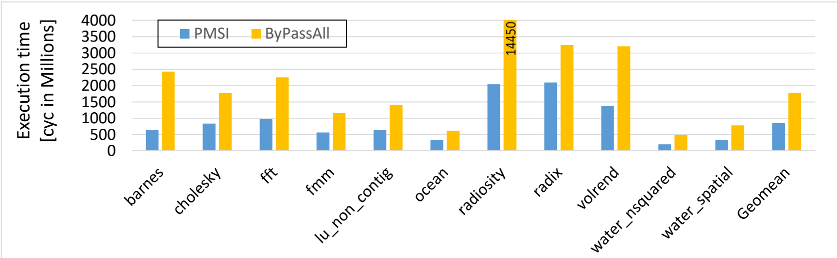


Figure 3 Execution time.

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

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

5.2 Per-Request WCL

Despite its average-case performance gains, PMSI suffers from large worst-case delays due to the introduced coherence interference. For instance, 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 L_{acc}^{miss} 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 $N \cdot L_{acc}^{miss}$. 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 [14]. From Table 2, total WCL of both bypassing and PMSI can be calculated as follows.

$$WCL_{perReq}^{PMSI} = 2 \cdot N^2 \cdot L_{acc}^{miss} + 2 \cdot N \cdot L_{acc}^{miss} + L_{acc}^{miss} \tag{1}$$

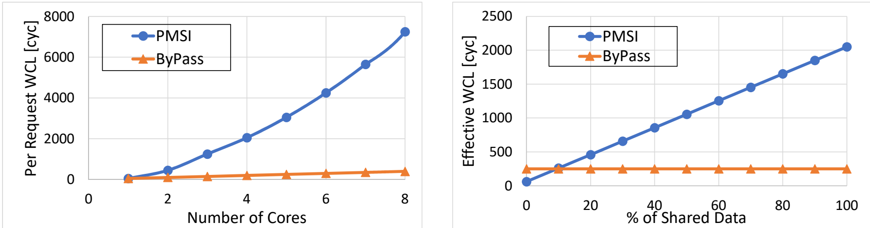
$$WCL_{perReq}^{ByPass} = N \cdot L_{acc}^{miss} + L_{acc}^{miss} \tag{2}$$

Figure 4a 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 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].

[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*.



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

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

Figure 4 Per-request WCLs (Equations 1 and 2) and effective WCLs (Equation 5.

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:

$$WCL_{tot}^{Bypass} = M \cdot WCL_{perReq}^{Bypass} = M \cdot L_{acc}^{miss} \cdot (N+1) \tag{3}$$

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 $M_{hits}^{private}$ and $M_{misses}^{private}$, 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 $L_{acc}^{private}$ and recall that the WCL to access a shared cache line (which includes coherence interference if exists) is WCL_{perReq}^{PMSI} as calculated by Equation 1. Accordingly, the cumulative total worst-case memory latency suffered by the task, WCL_{tot}, can be computed as:

$$WCL_{tot}^{PMSI} = M_{hits}^{private} \cdot L_{acc}^{private} + M_{misses}^{private} \cdot (N+1) \cdot L_{acc}^{miss} + M^{shared} \cdot WCL_{perReq}^{PMSI} \tag{4}$$

Dividing Equation 4 by the total number of task requests, we get the effective WCL of a single request (WCL_{eff}) as in Equation 5, which can be considered as the average WCL suffered by a single request to the cache.

$$WCL_{eff} = \%M_{hits}^{private} \cdot L_{acc}^{private} + \%M_{misses}^{private} \cdot (N+1) \cdot L_{acc}^{miss} + \%M^{shared} \cdot WCL_{perReq}^{PMSI} \tag{5}$$

To visualize this effect, Figure 4b plots the effective WCL for both bypass and PMSI for different percentage of accesses to shared data. Figure 4b 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_{perReq}^{shared} 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.

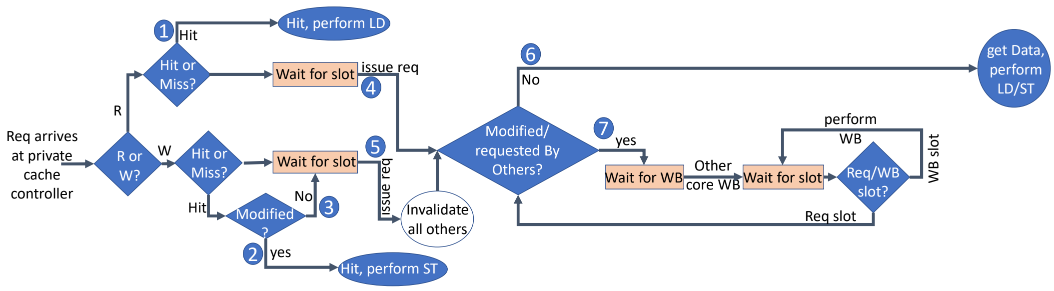


Figure 5 PMSI flow diagram.

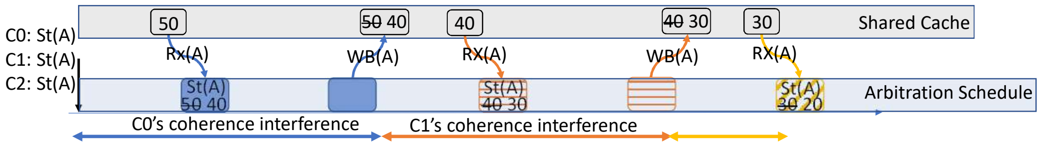


Figure 6 Coherence interference in case of writes for PMSI. C2 is the core under analysis and it has to wait for both C0 and C1 before it gains access to block A.

5.4 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 5. Figure 5 follows the design guidelines of PMSI [14].

5.4.1 Hit Scenario

In case of a read hit (shown as event ① in Figure 5) 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.

5.4.2 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 5 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 5. 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 5. As Figure 5 illustrates, the scenario at ⑦ is the one that triggers coherence interference and causes the largest delays.

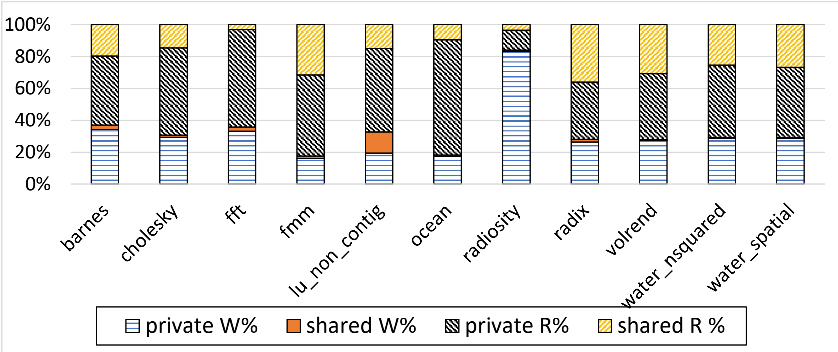


Figure 7 Breakdown of Splash benchmark memory requests.

5.4.3 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 6 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 6, 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 7 illustrates. The same observation holds for other benchmarks as well. For instance, we find that the PARSEC benchmark [5] suite and the EEMBC-auto [33] suite have on average 21% and 32% writes per application, respectively.

6 Proposed Solution

Motivated by the observations we made in Section 5, we propose DISCO: a discriminative coherence approach. The key idea behind DISCO is to eliminate by design the worst-case scenarios covered in the previous section, and therefore, avoid its significant coherence delays. On the other hand, DISCO still maintains a high average-case performance by employing coherence and enabling simultaneous access to shared data. These two 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

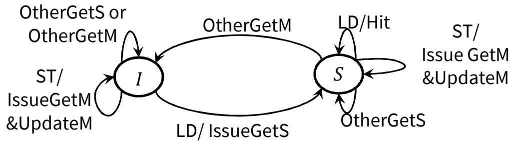
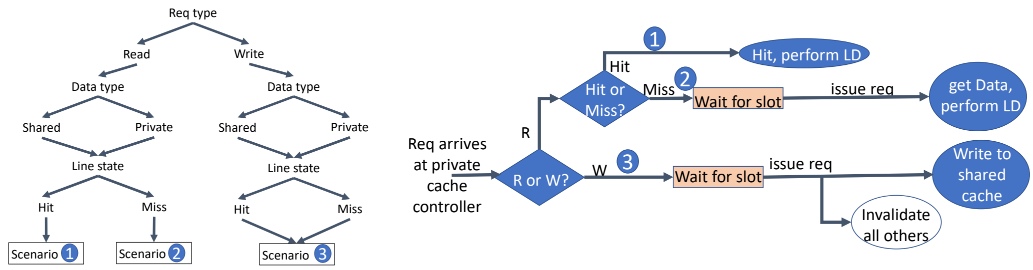


Figure 8 Coherence states of the simple SI protocol adopted by DISCO.



(a) Handling of different access scenarios.

(b) Flowchart of operations.

Figure 9 Proposed DISCO-AllW coherence approach.

needed because of data being updated privately by other cores such as in PMSI [14], and reduces the coherence protocol to the simple SI (or sometimes referred to as VI) protocol [38]. Figure 8 shows the coherence states of this SI protocol. It is worth noting that there are two different ways to realize DISCO, either by 1) implementing this SI 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 7.3.3. As Figure 8 illustrates, the M state is completely removed since no core will have a cache line modified in its private cache without updating the shared cache.

If a core has a read request to a cache line that is in the I state in its private cache, it issues a GetS message once granted access to the bus and moves to the S state once it receives data from the shared cache. In contrast, 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. If a core has a read request to a cache line that is in S state in its private cache, it is a hit and no change in the state is required. However, if the core has a write request to a line that is in the S 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. If while in the S state, a core observes an $OtherGetS$ message on the bus, it remains in the S state since the other core is requesting this line for a read and is not going to modify it. Contrarily, if the core observes an $OtherGetM$ message to a line it has in the S state, it has to invalidate its copy since the data is going to be updated by the other core.

Leveraging this simple 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 5 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 9 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. For example, one request can be a read hit to a private cache line, while another request could be a write hit to a shared cache line, etc. DISCO-AllW operates on these cases based on the following four rules:

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

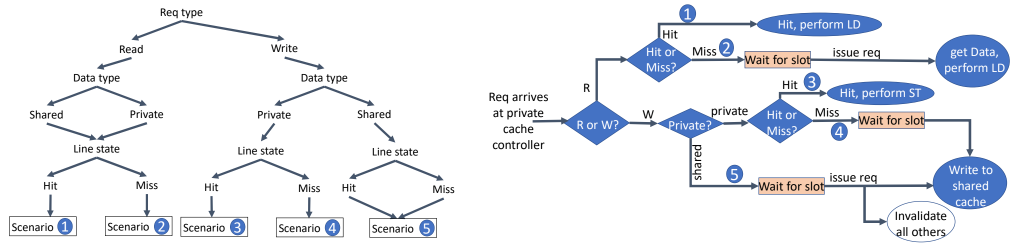
(A) It does not distinguish between shared and private lines, both are treated equally.
(B) It treats all writes equally by sending them to the shared cache.
(C) Read hits are allowed and can proceed without requesting an access to the shared bus.
(D) Read misses have to wait for an access to the shared 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 9a. Figure 9b 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 9b 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 which lines are shared, which facilitates its adoption if such knowledge



(a) Handling of different access scenarios. **(b)** Flowchart of operations.

Figure 10 Proposed DISCO-SharedW coherence approach.

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 10 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 10a.

1. **Scenario 1: A Read Hit in the Private Cache.**
2. **Scenario 2: A Read Miss in the Private Cache.**
   As illustrated in Figure 10 (compared to Figure 9), DISCO-SharedW handles Scenarios 1 and 2 exactly the same as DISCO-AllW.
3. **Scenario 3: 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 4: 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 5: A write to a shared cache line.** Write hits to shared cache lines are still not allowed. This is necessary 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)).

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. Excessive coherence delays because of multiple data transfers as discussed in Section 5 are completely eliminated.

▶ **Lemma 1.** *A request to a cache hierarchy deploying either versions of* DISCO *encounters a latency that is at most:*

$$WCL_{perReq}^{DISCO} = WCL_{perReq}^{DISCO\text{-}AllW} = WCL_{perReq}^{DISCO\text{-}SharedW} = N \cdot L_{acc}^{miss} + L_{acc}^{miss} \tag{6}$$

Proof. The proof directly follows from the fact that under the deployed TDM arbitration, a core in the worst case has to wait for all other cores before it can send an access to the shared bus. Recall that we have N cores and that the slot width is L_{acc}^{miss}. Thus, the worst-case arbitration latency a memory request can suffer is $N \cdot L_{acc}^{miss}$. In addition, as per definition, a request to the shared memory consumes an access latency of L_{acc}^{miss}. ◀

7.2 Total Worst-Case 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 2.** *Total worst case memory latency incurred by any task under* DISCO-AllW *can be calculated as:*

$$WCL_{tot}^{DISCO\text{-}AllW} = R_{hits}^{private} \cdot L_{acc}^{private} + (R_{misses}^{private} + R^{shared} + W) \cdot WCL_{perReq}^{DISCO} \tag{7}$$

Proof. Based on the discussion of DISCO-AllW in Section 6.1, we prove Lemma 2 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 1 and each of them can have a WCL of WCL_{perReq}^{DISCO}.

For the read requests to shared lines, denoted as R^{shared}: from Lemma 1, each one of those requests under DISCO-AllW suffers a WCL of WCL_{perReq}^{DISCO}. 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 1 with a WCL of WCL_{perReq}^{DISCO}. This constructs $WCL_{tot}^{DISCO\text{-}AllW}$ in Equation 7. ◀

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

$$WCL_{tot}^{DISCO\text{-}SharedW} = M_{hits}^{private} \cdot L_{acc}^{private} + (M_{misses}^{private} + M^{shared}) \cdot WCL_{perReq}^{DISCO} \tag{8}$$

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}^{DISCO} as per Lemma 1. Finally, Requests to shared lines, M^{shared}, also suffer the WCL of WCL_{perReq}^{DISCO} 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 8. ◀

7.3 Other Considerations: A discussion

In this section, we discuss factors that we believe are important to consider for the generalization of the proposed approaches.

7.3.1 On the Derivation of the Total WCL

Private and Shared Data. Equations 4, and 7–8, 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 [27]. 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:

$$WCL_{tot}^{PMSI} = M^{private} \cdot (N+1) \cdot L_{acc}^{miss} + M^{shared} \cdot WCL_{perReq}^{PMSI} \tag{9}$$

$$WCL_{tot}^{DISCO} = M \cdot WCL_{perReq}^{DISCO} \tag{10}$$

These two equations also can 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 PMSI nor DISCO. Per-request WCL also remains as previously calculated in Equations 1 and 6.

Reads and Writes. Another assumption that is made by Equation 7 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 [24]. Nonetheless, if such information is not available, Equation 10 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.

7.3.2 Effect of Write-backs Due to Replacement 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 TDM period due to this write back of the evicted line. This adds $N \cdot L_{acc}^{miss}$ cycles to the WCL_{perReq} in Equation 6 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 4 calculates the new total WCL under DISCO-SharedW, while accounting for the delay effect of the write-backs due to cache replacement.

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

$$\begin{aligned} WCL_{tot}^{DISCO\text{-}SharedW} &= M_{hits}^{private} \cdot L_{acc}^{private} + (M_{misses}^{private} + M^{shared}) \cdot WCL_{perReq}^{DISCO} \\ &+ N \cdot L_{acc}^{miss} \cdot WB \end{aligned} \tag{11}$$

Proof. The proof directly follows from the proof of Lemma 3, 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 can take a maximum of one TDM period to finish. This gives a total delay of $N \cdot L_{acc}^{miss} \cdot WB$, which is the last term in Equation 11. ◀

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 7 with 11, this is true for two reasons. 1) $WB \leq W^{private}$ as previously explained, and 2) $WCL_{PerReq}^{DISCO} > N \cdot L_{acc}^{miss}$.

7.3.3 Realization in Existing Architectures

One of the main advantages 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 in existing architectures. The first version of DISCO

(DISCO-AllW) requires only the ability to bypass private caches for write requests. Contrarily, the second version (DISCO-SharedW) requires, in addition to write bypassing, the ability to distinguish between shared and private lines. We discuss these two requirements below.

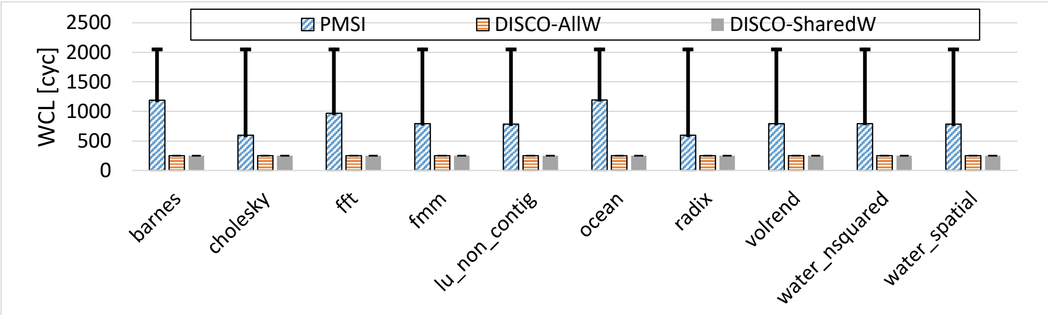
1) Selective Bypassing of Writes. Bypassing writes in the private caches can be realized in existing hardware by multiple means. First, a write-through cache achieves exactly the necessary behavior. Many existing architectures enable the user to set caches to operate as write-through caches. For instance, ARM allows the user to switch to write-through caches using a special register named Cache Behavior Override Register [2]. The same register also allows for setting caches as non-write-allocate, which means upon a write request, the cache line is written in the shared cache but is not fetched to the private cache. This is the same behavior we adopted to reduce the WCL. However, it is worth noting that as explained at the beginning of this section, DISCO can operate correctly even if this capability of non-write-allocate is not provided, albeit with two memory transfers per slot in the worst case. 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 [2], which is again exactly the same behavior needed for DISCO. Intel processor also provides various control registers to support setting caches to different cache types including write-through [21]; nonetheless, it seems to apply the setting for all cache levels, which forces writes to be sent to the main memory.

2) Isolation Between Shared and Private Data. Isolation between shared and private lines is needed only by the DISCO-SharedW. This can be achieved in existing hardware by placing the shared memory in specific memory regions and then handle requests to every cache line differently in DISCO-SharedW based on the address of this line (whether it is within the boundaries of the shared region or not). This is possible since the aforementioned support about write bypassing can be applied to only specific regions in the memory both in ARM's [2] and Intel's platforms [21]. For instance, DISCO-SharedW can set those shared regions to write-through, while private regions operate normally in a write-back fashion.

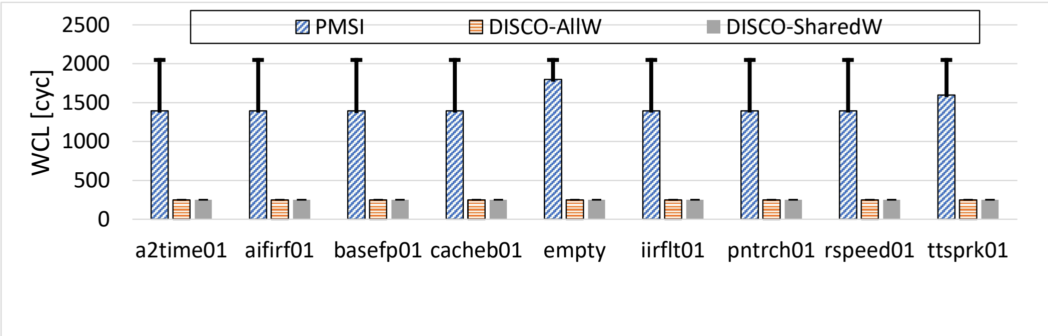
8 Evaluation

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 TDM arbiter. 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, simialr to existing solutions [14, 23], we set $L2$ to be a perfect cache, i.e. all requests to $L2$ are hits. It is worth noting that this setup has no effect on the evaluated approaches, while it allows to avoid the effect of off-chip memory interference on the total execution time. The DRAM overheads are considered additive to the latencies derived in this work and can be computed using existing work [12].

We deploy both versions of DISCO: DISCO-AllW and DISCO-SharedW. In addition, we also implement PMSI [14] and ByPassAll solutions discussed in Section 5. We use benchmarks from the SPLASH3 multi-threaded benchmark suite [33] as well as the EEMBC-Auto suite [33]. The simulation environment integrates with the Intel PIN tools [28] 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.



(a) Splash3.



(b) EEMBC.

Figure 11 Both analytical (T bars) and experimental (solid bars) per-request WCL.

8.1 Per-Request Worst-Case Latency

Figure 11 delineates the WCL for any request to the cache hierarchy in a four-core system for both SPLASH3 (Figure 11a) and EEMBC (Figure 11b). 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 is able to reduce 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 11a), 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 11a), the gap is more than 45% for most benchmarks. On the other hand, DISCO's analytical and experimental WCLs are identical, which indicates the tightness of the derived bounds. 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 11), 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 12 delineates the total WCL of all the evaluated approaches for the SPLASH-3 benchmarks. To facilitate readability, all results in Figure 12 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

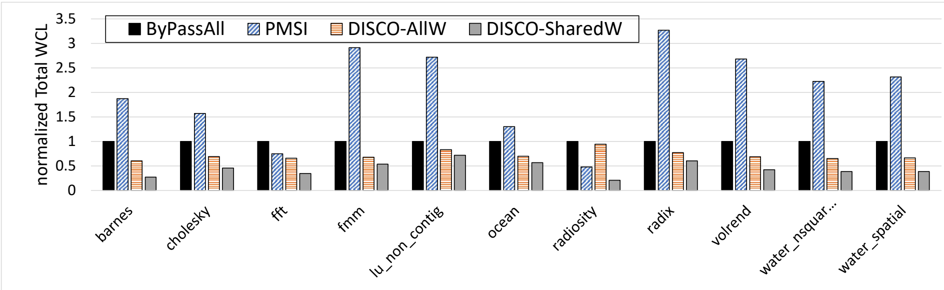


Figure 12 Total worst-case latency of Splash3.

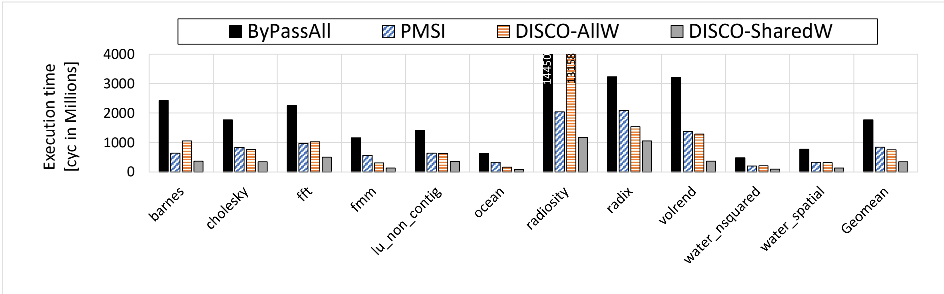
a time period *t* and is calculated in Equations 3, 4, 7, and 11 for ByPassAll, PMSI, DISCO-AllW, and DISCO-SharedW, respectively. Figure 12 introduces several interesting observations.

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 4, the term that suffers the maximum latency of WCL^{PMSI}_{perReq} 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 12. 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 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 as we highlighted in Section 8.1, 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 2-4, 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.

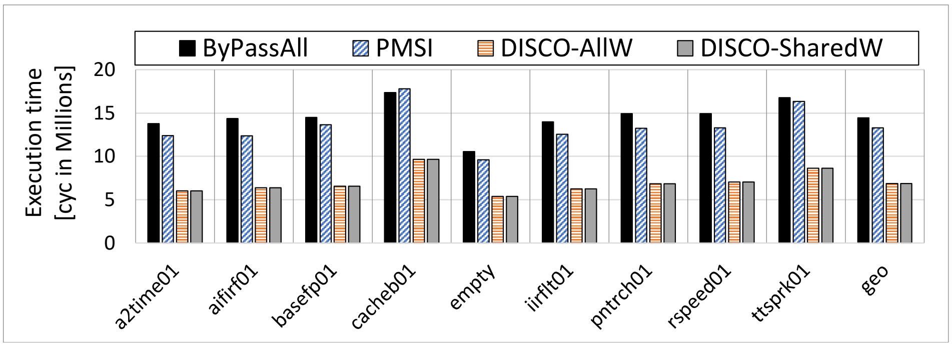
8.3 Average-Case Performance (Execution Time)

Figure 13 depicts the overall execution time for both SPLASH3 (Figure 13a) and EEMBC (Figure 13b) 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 for the SPLASH3 benchmarks. 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 for SPLASH3 benchmarks (up to 100% and 12% better performance on average). Nonetheless, PMSI has better



(a) Splash3.



(b) EEMBC.

Figure 13 Execution time.

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. 3) Even with the synthetic maximum-sharing scenario of EEMBC experiments (Figure 13b), PMSI outperforms ByPassAll, though slightly. In contrast, DISCO shows its maximum performance benefit with increased sharing for two main reasons. a) On the one hand, it does not suffer from the large coherence interference delays incurred by PMSI due to writes. b) On the other hand, it does not suffer from large delays due to forcing all requests to access L2 incurred by ByPassAll as DISCO allows read hits in the private caches. Please note that both versions of DISCO incur exactly same behavior under the EEMBC experiment since all requests (including writes) are shared among all cores. 4) Figure 13a 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 design choice.

8.4 Average-Case Performance (Average-Case 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 14. Figure 14 confirms the same behavior observed in the execution time in Figure 13a. 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 $12\times$

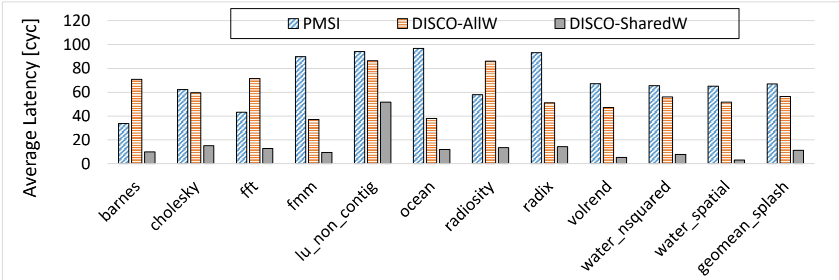


Figure 14 Average latency of Splash3.

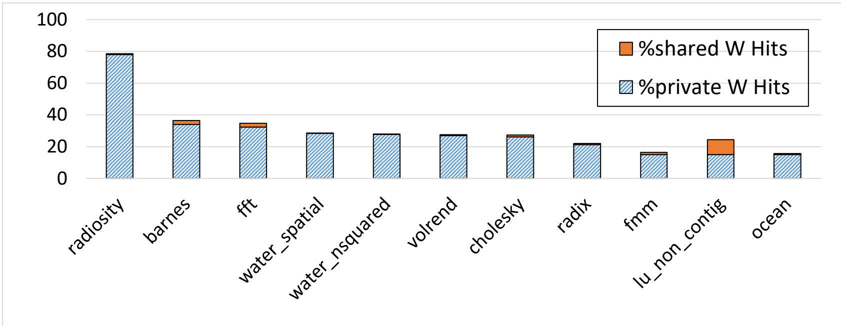


Figure 15 Measured PMSI write hits for Splash3.

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. Figure 15 plots write hit both to shared and private lines as a percentage from the total number of issued requests. Benchmarks are shown in a decreasing order in number of write hits to private lines. Figure 15 confirms 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.

9 Conclusion

Modern real-time systems applications mandate data sharing. In this paper, 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. DISCO provides the tight latency bounds by eliminating the scenarios that cause high coherence interference under traditional coherence protocols. DISCO can be realized in systems that support write through caches or cache bypassing without any modifications. If such support is not available, it can be realized by modifying the cache controller to adopt the coherence protocol. 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 Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable SDRAM memory controller. In *IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ ISSS)*, 2007.

2 ARM. ARM arm1176jz-s technical reference manual. 2013.

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

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

5 Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

6 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.

7 B. Cilku, B. Frömel, and P. Puschner. A dual-layer bus arbiter for mixed-criticality systems with hypervisors. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*, pages 147–151, July 2014. doi:10.1109/INDIN.2014.6945499.

8 Leonardo Ecco and Rolf Ernst. Improved dram timing bounds for real-time dram controllers with read/write bundling. In *2015 IEEE Real-Time Systems Symposium*, pages 53–64. IEEE, 2015.

9 Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTAS)*, 2014.

10 Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 2015.

11 Giovani Gracioli and Antônio Augusto Fröhlich. On the design and evaluation of a real-time operating system for cache-coherent multicore architectures. *ACM SIGOPS Oper. Syst. Rev.*, 2015.

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

13 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.

14 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.

15 M. Hassan and H. Patel. Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

16 Mohamed Hassan. Heterogeneous mpsocs for mixed-criticality systems: Challenges and opportunities. *IEEE Design & Test*, 2018.

17 Mohamed Hassan and Hiren Patel. A framework for scheduling DRAM accesses for multi-core mixed-time critical systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

18 Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. PMC: A requirement-aware DRAM controller for multicore mixed criticality systems. *ACM Trans. Embed. Comput. Syst.*, 2017.

19 Farouk Hebbache, Mathieu Jan, Florian Brandner, and Laurent Pautet. Shedding the shackles of time-division multiplexing. In *IEEE Real-Time Systems Symposium (RTSS)*, 2018.

20 John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

21 Intel. Intel 64 and IA-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part*, 1(64), 64.

22 Javier Jalle, Eduardo Quinones, Jaume Abella, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In *IEEE Real-Time Systems Symposium (RTSS)*, 2014.

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

24 Sung-Kwan Kim, Sang Lyul Min, and Rhan Ha. Efficient worst case timing analysis of data caching. In *Proceedings Real-Time Technology and Applications*, pages 230–240. IEEE, 1996.

25 NG Chetan Kumar, Sudhanshu Vyas, Ron K Cytron, Christopher D Gill, Joseph Zambreno, and Phillip H Jones. Cache design for mixed criticality real-time systems. In *IEEE International Conference on Computer Design (ICCD)*, 2014.

26 Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures. In *International Conference on Real-Time and Network Systems*, 2010.

27 Benjamin Lesage, Isabelle Puaut, and André 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.

28 Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

29 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54. IEEE, 2013.

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

31 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ACM Annual International Symposium on Computer Architecture (ISCA)*, 2009.

32 Marco Paolieri, Eduardo Quiñones, Fransisco J. Cazorla, and Mateo Valero. An analyzable memory controller for hard real-time CMPs. *Embedded System Letters (ESL)*, 1:86–90, 2009.

33 Jason Poovey et al. Characterization of the EEMBC benchmark suite. *North Carolina State University*, 2007.

34 Jan Reineke, Isaac Liu, Hiren D Patel, Sungjun Kim, and Edward A Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ ISSS)*, 2011.

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

36 Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. Towards time-predictable data caches for chip-multiprocessors. In *Springer International Workshop on Software Technolgies for Embedded and Ubiquitous Systems (IFIP)*, 2009.

37 Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russel Kegley, Dennis Perlman, Greg Arundale, et al. Single core equivalent virtual machines for hard real—time computing on multicore processors. Technical report, 2014.

38 Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 2011.

39 N. Sritharan, A. M. Kaushik, M. Hassan, and H. Patel. Hourglass: Predictable time-based cache coherence protocol for dual-critical multi-core systems. 2017.

40 Nivedita Sritharan, Anirudh Mohan Kaushik, Mohamed Hassan, and Hiren Patel. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. 2019.

41 Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 1990.

42 Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *ACM Annual Design Automation Conference (DAC)*, 2008.

43 B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Making shared caches more predictable on multicore platforms. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

44 Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009.