Shared Data Kills Real-Time Cache Analysis. How to Resurrect It?

Safin Bayes
Electrical and Computer Engineering
McMaster University
Hamilton, Canada
bayess@mcmaster.ca

Mohamed Hossam
Electrical and Computer Engineering
McMaster University
Hamilton, Canada
mohamed.hossam@mcmaster.ca

Mohamed Hassan
Electrical and Computer Engineering
McMaster University
Hamilton, Canada
mohamed.hassan@mcmaster.ca

***Abstract*—While data sharing is becoming a necessity in modern multi-core real-time systems, it complicates system analyzability and leads to significantly pessimistic latency bounds. This work is a step towards facilitating high-performance and coherent data sharing in real-time systems by tackling two main problems. The first is a well-acknowledged one: shared caches render cache analysis techniques useless and all cache accesses have to be assumed *misses*. The second is a new one, where we show that coherence interference voids classical cache analysis techniques. We contribute a solution that tackles both problems by leveraging time-based cache coherence and a novel methodology to integrate its effect into cache analysis. Thanks to this solution, we enable the usage of shared memory hierarchy with coherent shared data, while we prove that we are able to restore cache analysis; and hence, provide much tighter memory latency bounds.**

***Index Terms*—coherence, cache, analysis, multi-core, shared data**

I. Introduction

Timing Predictability is a must for real-time systems, which is usually ensured by obtaining an upper bound of the worst-case execution time (WCET) of each task. For obtaining the WCET, usually the task is thoroughly examined in-isolation (i.e. without considering any other task) either statically through static analysis techniques or experimentally through exhaustive testing [1]. Part of this analysis is concerned with the cache behavior of the task. Caches are generally problematic for real-time analysis due to their significant latency variability based on whether a request hits or misses in a particular cache level. In multi-core systems, the in-isolation analysis is no longer sufficient since the WCET of a task running in one core will depend on the behavior of tasks running on the other cores as cores compete on accessing shared resources in the platform.

With shared caches, for example, *an interfering core can evict a cache line of another (victim) core which is mapped to the same shared cache set*. With inclusive caches where the cache lines in the shared cache form a superset of those in the private caches, *this victim cache line has to be evicted from the private cache(s) of the victim core as well*. We denote this as ***Problem 1***, and we illustrate this in Figure 1(a). Problem 1 clearly voids the aforementioned in-isolation cache analysis since every single access of the victim core could be subjected to such interference, and thus, has to be assumed a miss. To address this problem, cache partitioning [2] has been a common approach, where each core is assigned one or more private partitions of the shared cache. Unfortunately, this approach only applies to independent tasks since it prohibits any communication (shared data) between tasks in different partitions. Therefore, recent works focused on enabling data-sharing in real-time systems using cache coherence techniques either by proposing new protocols that ensure predictability by design [3], [4], [5], devising techniques to adopt standard commodity coherence protocols in real-time systems [6], [7], [8], or analyzing existing cache-coherent multi-core platforms targeting real-time systems such as avionics [9], [10]. However, we observe that all these coherence works focus on deriving and tightening the worst-case latency (WCL) that a single memory request suffers upon accessing a coherent cache subsystem. Despite the importance of minimizing per-request WCL, we observe a source of interference due to coherence which voids the in-isolation cache analysis. We denote this source as ***Problem 2*** and articulate it as follows: *Under cache coherency, an interfering core can request a cache line that resides in the victim core's private cache, which the victim core has to write back or invalidate*. Figure 1(b) illustrates this problem. In worst-case, every access from the victim core is subjected to an interference due to Problem 1 or Problem 2. The total worst-case memory latency (WCML) for a core c_i with a worst-case total number of memory requests denoted as M_i^{tot} can be bounded by Equation 1, where L^{hit} is the *hit* latency, WCL_i^{miss} is the maximum latency for a *miss* in the private cache, and M_i^{hit} and M_i^{miss} are the total number of private cache *hits* and *misses* occurred in the task, respectively and $M_i^{hit} + M_i^{miss} = M_i^{tot}$. Since every request can suffer interference that results in evictions due to Problems 1 or 2, existing predictable cache coherence works unanimously assume every request as a miss to the private cache ($M^{hit} = 0$, $M^{miss} = M^{tot}$). This yields a safe, but an overly pessimistic bound for the WCML.

$$WCML_i = M_i^{hit} \cdot L^{hit} + M_i^{miss} \cdot WCL_i^{miss} \quad (1)$$

To address both problems, we propose a novel approach which enables us to leverage shared coherent cache hierarchies of modern multi-core platforms while still being able to apply cache analysis techniques and use hit/miss classification to tighten the task's WCET. Unlike existing works [3], [5], we do not propose a completely new coherence protocol. Instead,

979-8-3503-4859-0/DATE24/© 2024 EDAA

we capitalize on a property of a class of cache coherent protocols known as time-based coherence, which was used to optimize coherence traffic in modern multi-core systems [11], GPUs [12], and was recently proposed for real-time systems [5] as well. In time-based coherence, once a core obtains a cache line in its private cache, it entertains guaranteed hits to this line for a certain period of time. The main aspect of this paper is to show how we integrate this property with in-isolation analysis to restore the latter, which enables the derivation of much tighter (with an order of magnitude in some scenarios) WCML bounds.

II. Cache Coherence Background

Cache coherence is the de facto standard in modern Commercial-Off-The-Shelf (COTS) platforms to ensure the correctness of shared data while allowing cores to have simultaneous access to this data. This is possible by enforcing a set of rules on the cores dictated by a cache coherence protocol. We use the standard MSI protocol to explain the basics of cache coherency. The MSI protocol consists of three states: Modified (M), Shared (S), and Invalid (I). The M state denotes that the core has modified the data in the cache line, and has read and write permissions on it. The S state indicates the cache line was read, but not modified. Cores have only read permission on lines in S state. Finally, the I state means that the core does not have a valid copy of the cache line. Multiple cores can have the same cache line in S state simultaneously, but only one core can have a cache line in M state. This principle is referred to as Single Writer Multiple Readers (SWMR) invariant. Cores transition to different states either by issuing or responding to coherence messages. To obtain a cache line in M (or S), cores issue a GetM (GetS) message. To evict a dirty line from the cache, cores issue a PutM message. Depending on their states, other cores must respond to this message to enforce SWMR.

Transient States. In addition to these states known as stable states, there are transient states that represent transitions between stable states. For example, when a core issues a GetM (GetS) message, it moves to a state, IM^a (IS^a), which indicates the transition between I and M (S) states. The superscript a indicates that the core is waiting for the coherence message to appear on the shared bus, following which, it moves to IM^d (IS^d), where it waits to receive the data. When the core obtains the data, it completes its transition to M (S) state.

Time-Based Coherence. Unlike conventional protocols, time-based coherence does not necessitate an immediate response to coherence messages on the bus. Once the core obtains a copy of the data in its private cache, it will maintain the validity of the data for a certain amount of time. Only after the expiration of the timer, the core may invalidate the cache line in response to other coherence messages. If there are no messages for that particular cache line, the timer will replenish, and the core will keep the data for another period of the timer. In addition to the transient states introduced above, time-based protocols consist of timer-related states, such as, M^TI and S^TI [5]. The M^TI state indicates that the core will write back the data after the timer expires, while S^TI denotes that the core will invalidate its shared copy after the timer expires.

A. Motivation

Figure 1a illustrates Problem 1. First, core c_1 issues a request to a cache line, E, which does not exist in its L1 nor in the shared cache. To bring in E to the shared cache, cache line A is chosen to be evicted ❻. Assuming that the shared cache is inclusive, A is also evicted from c_0's L1. As a result, when c_0 issues a request to A later at ❾, the request misses in its L1 and the shared cache. In Figure 1c, we illustrate how time based coherency prevents this miss. After A is selected as the replacement candidate ❺ following c_1's request ❶, A is not evicted because c_0's timer is still running. Subsequently, c_0's request to A ❼ hits in the L1 which instead missed in Figure 1a. Only when the timer expires ❽, A is evicted from c_0's L1 and the shared cache to bring in E for c_1.

Now we illustrate Problem 2 in Figure 1b. First, c_1 issues a request to A ❶. c_0 has A in M state in its L1, but due to the SWMR principle, it must invalidates its copy of A and send the data to c_1. Consequently, c_0's request to A ❻, misses in the L1. However, with time-based coherence in Figure 1d, c_0 does not invalidate A immediately; instead, it moves A to M^TI state, indicating that it will invalidate its copy after the timer expires. As the timer is still running at ❺, c_0's request to A hits in the L1. After the timer expires, c_0 sends the data to c_1 ❼.

III. System Model

We consider a multi-core system composed of a set of N cores defined as $\mathcal{N} = \{c_i | N > i \geq 0\}$. Cores can execute instructions using In-Order or Out-of-Order (OoO) pipelines. We do not consider any limit on the number of tasks a core can run. Instead, we are interested in bounding the processing delay of an individual memory access, and aggregate it over a period of time (and a number of requests) that can span a task or more executing on that core.

The cores have a private cache (L1) and access to a shared on-chip Last Level Cache (LLC), as well as an off-chip DRAM upon missing in the LLC. The caches are non-blocking and allow *cache hits to be serviced even if there are pending misses in the cache*. This is usually referred to as a *hit-over-a miss*. However, similar to existing work [7], in our analysis, we assume that each core can have only one L1 miss request being serviced by the shared cache at any given time (in other words, it does not allow for *miss-over-a miss*). This allows our system to leverage some of the performance improvement from OoO pipelines whilst maintaining predictability. While cache inclusivity does not matter for Problem 2, we consider an inclusive cache system to account for Problem 1.

The L1 caches communicate with each other and the shared memory through a shared bus. Existing works employed both unified buses [3], [8], [6], [5] and split buses [7]. While our solution is independent of the bus architecture, we consider a unified bus for the analysis. The bus allows data transfer between an L1 cache and the shared memory, and between different L1 caches. Cores are granted access to the bus based on a predictable arbitration mechanism. The coherency

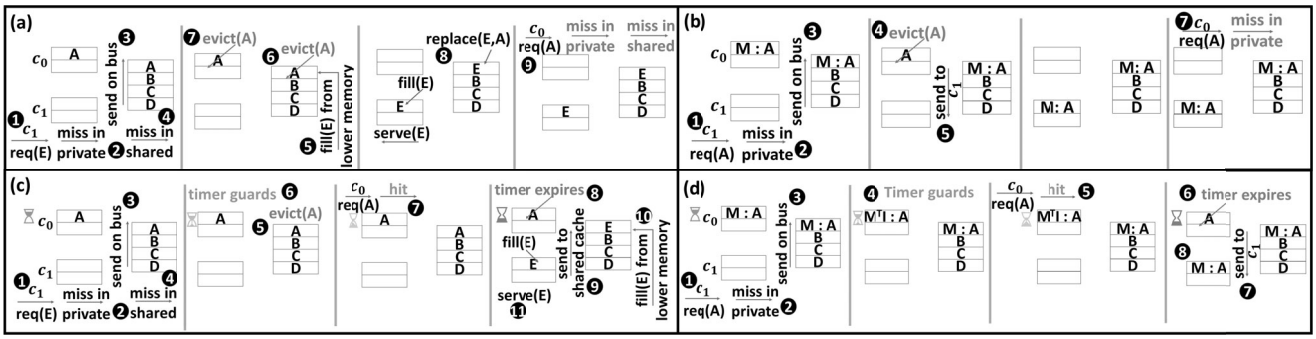

Fig. 1: Illustration and mitigation of (a) (c) Problem 1 and (b) (d) Problem 2.

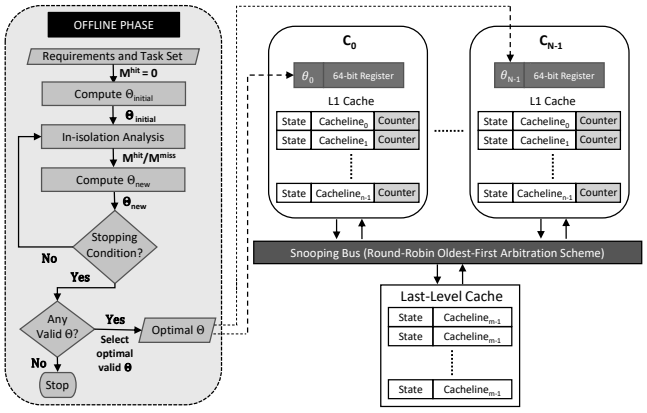

Fig. 2: Diagram of the proposed solution. The colored blocks indicate the proposed modifications to the system architecture.

of shared data is maintained through a time-based coherence protocol, which we discuss in more details in Section IV-D.

IV. Proposed Solution

This work aims to preserve the in-isolation analysis of tasks using time-based coherency, so that we can guarantee a number of hits for the task under analysis, and tighten its WCML bound. Towards this goal, we propose a methodology to integrate time-based coherence with the in-isolation cache analysis of the tasks. This is highlighted in the left side of Figure 2. Then, we present PENDULUM*, the architecture to support the time-based protocol described in the right side of Figure 2. The architecture and the coherence protocol extends PENDULUM [5] in several ways that we discuss in the following subsections.

A. System Architecture

Cache Controller. In addition to the coherence states, two timer-specific components are needed. 1) One register per core which holds the timer threshold value. This register is configurable and can be programmed by the software layer to set the desired timer value. 2) One 16-bit counter per cache line. Once a cache line is fetched to a core's private cache, the counter will be set to the timer register value. Then, the counter will be decremented by one in every cycle until it reaches 0. We find 16-bit for the registers and the counters to be sufficient. This adds only around 3% overhead for a 64B cache line.

Shared Bus and Arbitration. The proposed solution is independent of the specifics of the bus arbitration as far as it is predictable, and thus, should work seemingly with any predictable bus arbitration. That said, we adopt in our implementation the Round-Robin Oldest-First (RROF) arbitration scheme [13]. The RROF mechanism is similar to the traditional Round-Robin (RR) mechanism in that both maintain a cyclic order of cores with ready requests. In RR, the core loses its order after it gets access to the bus. In RROF, however, the core loses its order only after its oldest request has been served. This characteristic of RROF allows us to obtain a tighter bound for WCL^{miss}, as we will discuss at the end of Section IV-C. Figure 3 shows an illustrative example for the operation of PENDULUM*. Initially ❶, all cores issue a write request to cache line A. The cyclic order of the cores and their respective request queues are also shown in Figure 3. At ❶, since c_0 is at the top of the RROF order, it issues the request, receives the data, and starts its timer θ_0. As c_0's oldest request is served at ❶, it loses its position in the RROF order. Also, c_1 issues its request at ❶, but does not lose its RROF order because it has to wait for θ_0 to expire before c_0 can send the data to c_1. Similarly, at ❷, c_2 issues its request, and waits for c_1 to send the data. At ❸, θ_0 expires just when c_0 is serving a request to X_0. Once it is served, c_1 gets access to the bus ❹ as it is at the top of the RROF order with ready data. c_0 sends the data to c_1 after which θ_1 starts. c_1 also loses its RROF order as its oldest request is served. Then, θ_1 expires just when c_1 is issuing a request to X_1 ❺. Now, as c_2 is at the top of the RROF order, it gets access to the bus and acquires A from c_1 ❻.

B. Worst-Case Per-Request Memory Latency

Lemma 1: (***Per-Request WCL,*** WCL_i^{miss}) For the proposed system, the maximum WCL a request by core c_i suffers can be calculated by Equation 2, where SW (Slot Width) is the maximum bus access time a core is allowed at once and θ_i represents the timer threshold for c_i.

$$WCL_i^{miss} = SW + (\sum_{\substack{j=0 \\ j \neq i}}^{N-1} \theta_j) + 2 \cdot (N-1) \cdot SW \quad (2)$$

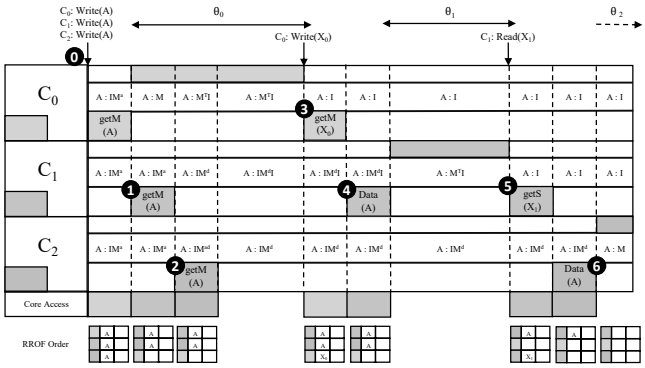

Fig. 3: An example of PENDULUM* operation.

Proof: The worst-case scenario for any request, $r_i(A)$ from core c_i to a cache line A on the shared memory bus occurs when this request is broadcasted directly after all other cores have already issued a store request to A one after the other. In this case, c_i has to wait for the following to happen before $r_i(A)$ can be serviced. 1) Each other core c_j will fetch A to its private cache (according to the broadcasting order), 2) serve its write request $r_j(A)$ and keep A for θ_j before finally 3) sending the updated data to the next core in the broadcast order. We now compute each of these components. First, $r_i(A)$ waits in worst for SW cycles before the first core receives the data from the shared cache. This is the first term in Equation 2. Except the first one, each core waits for its sender's timer, θ_j, to expire before the sender can initiate data transfer. (Second term in Equation 2). The timer of each sender might expire in the middle of a slot (1 cycle after the beginning of a slot in worst-case). So the owner core must wait for SW cycles (in worst-case) before it can initiate data transfer to the next requester. Here, we highlight the significance of adopting the RROF arbiter. In our system, data transfer takes place when receiver has access to the bus. With RR arbiter, we cannot guarantee that the next slot after the expiration of sender's timer will be granted to the next receiver in the broadcasting order. In worst-case, the next receiver will be granted access after $(N-1)$ SWs. However, recall that in RROF, cores do not lose their order in the arbitration queue if the oldest request is not serviced. Since our system permits only one miss request to be serviced, the store request to A is the oldest request for all the cores, and hence the next receiver is guaranteed the slot following the expiration of the sender's timer. Finally, it takes an additional SW cycles to transfer the data from the owner core to the target core. Therefore, it takes in worst-case $2 \cdot SW$ cycles to transfer data from a core to another core after the timer expiration. This justifies the last term in Equation 2. ■

C. Integrating Timers into Cache Analysis

High-Level Intuition and Steps. Consider Θ is the set of coherence timer registers and is denoted as $\Theta = \{\theta_i | \forall i \in [0, N-1]\}$, where $\theta_i \in \{\mathbb{Z}^{0+}\}$ and $\mathbb{Z}^{0+}$ is the set of non-negative integers. θ_i represents the timer constant threshold for a core c_i. Under the operation of time based coherence, once c_i **receives the data of a cache line L in its private cache, it entertains a guaranteed hit access to L for at least θ_i cycles regardless of the activity of other interfering cores**. We say at least because once the timer expires, it can be replenished if other cores do not require c_i to give up L. We leverage this fact by conducting the in-isolation task analysis using a certain θ_i configuration that we honour during the co-runner (interference) case. As a result, the obtained number of hits using this θ_i value from the in-isolation analysis represents a lower (and hence a safe) bound on the number of hits that c_i will entertain under the interference scenario. Whether c_i replenishes the timer of a certain cache line at the end of its timer expiration depends on the behavior of other cores, so we run the in-isolation analysis with the assumption that when the timer expires, it will not be replenished and the line is invalidated. This is a must to ensure that the obtained hit count from the in-isolation analysis remains a safe lower bound regardless of the behavior of the contending cores in the interference scenario. In summary, we conduct the following steps. 1) We conduct the tasks' in-isolation cache analysis with a certain configured Θ such that once a core c_i obtains a cache line L in its private cache, it keeps it for θ_i cycles during which all accesses to L will be hits. 2) After the timer expires, the core invalidates L. 3) This gives us a number of hits that are guaranteed regardless of the contention setup as long as c_i uses the same θ_i value as the one used in isolation.

Obtaining Timer Values. The question now is: *how to set the timer values of different cores?* In the light of Equation 1, the timer values are subjected to a trade-off. Increasing the timer allows the owner core to protect its cache line from evictions due to other cores' requests; potentially leading to higher M^{hit} and lower M^{miss}. On the other hand, it also entails that other cores have to wait more for the owner core to give up the cache line, which increases the miss penalty, WCL^{miss}. While the Θ–WCL^{miss} relationship can be formally described in a closed-form (which we derive in Section IV-B), capturing the Θ–M^{hit} relationship is not as straightforward as it is dependent on the application's memory behavior. To determine Θ, we therefore, propose a fixed-point search algorithm as follows.

Fixed-Point Iterative Search Engine. We implement a search engine to obtain an appropriate set of timers based on the fixed point iteration method described by the flowchart in Figure 2. First, we assume all requests are misses (similar to existing works, i.e. $M^{hit} = 0$) to obtain the maximum upper bound on WCML for each task. We use this bound to obtain the initial set of timers, $\Theta_{initial}$ by solving a system of N equations of Equation 1 (one per core). Then we start the fixed point iteration technique where at every iteration, we analyse the memory requests of the tasks in isolation with θ obtained from the previous iteration (In the first iteration, the cores use θ from $\Theta_{initial}$) to obtain a safe M^{hit} and M^{miss}. For each task, we represent its required WCML in Equation 1 with M^{hit} and M^{miss} obtained from isolation analysis and solve the system of Equations 1 to obtain the new set of timers, Θ_{new}. We repeat these steps until the number of hits converges (i.e. the change

in the number of hits between two successive iterations does not exceed a pre-defined threshold). Once out of the search, if there are valid sets of timers, we pick the best Θ: the set of timers which results in the minimum WCML averaged for all cores ($(\sum_{i=0}^{N} WCML_i)/N$). If there are no such valid sets, the system is deemed unschedulable.

In-Isolation Analysis. This approach can be used with any cache analysis technique (either analytically or experimentally). For sake of illustration in this paper and without loss of generality, we build a model for the cache hierarchy with the time-based coherence. In this model, we maintain the cache lines with non-expired timers to a buffer that we name *active buffer*. 1) If the memory instruction is to a cache line present in the active buffer, we classify it as a hit. 2) If the request is a miss, we place it in another buffer that we denote as *pending buffer* for WCL^{miss} cycles (Derived in Lemma 1). This implies that every miss request is serviced after it incurs the worst possible delay. Although this is a pessimistic assumption, it is a necessity for safe analysis. 3) After WCL^{miss} cycles, the requested cache line is placed in the active buffer and it remains there until the expiration of the timer.

D. PENDULUM* *vs* PENDULUM

We now highlight the key extensions introduced in PENDULUM*.

1) **Arbiter.** Unlike PENDULUM which employs traditional Time-Division-Multiplexing (TDM) scheme for arbitration, we chose to use RROF [13]. This decision brings improvement not only on the WCL but also on performance. Thanks to RROF, PENDULUM* **has a per-request WCL that is linear in the number of cores (proof is in Section IV-B) instead of quadratic bounds provided by** PENDULUM (Lemma 2 in [5]). For performance, RROF is a dynamic RR-based arbiter and hence does not suffer from the known idle slots problem of traditional TDM.

2) **Timers.** We make two changes to the timers compared to those in PENDULUM. a) Instead of using two counters per cache line as in PENDULUM, we use only one. The owner core in PENDULUM uses two counters based on the requester core's criticality (critical vs non-critical). Since the owner invalidates the cache line at the earliest among both timers, we find it functionally sufficient to keep only one counter that tracks the earliest expiration time to resemble PENDULUM's operation. b) PENDULUM mandates the timer threshold values to be multiple of TDM periods to minimize the number of bits for the counters. However, we find this limitation to be tightly-coupled to traditional TDM since the period has a constant width regardless of the system's activity and does not really work for dynamic arbiters. In addition, our experimentation revealed that mandating the timer to be only multiples of TDM period prevents the system from setting the timer to optimal values both from WCL and performance perspectives. Therefore, we chose to let the timer threshold to be of any non-negative integer value.

3) **Coherence Protocol.** The PENDULUM state machine [5], similar to other existing works [3], [4], is presented for the cache of a perfect LLC (i.e., no misses to the shared LLC). We extend the state machine to account for the additional states, events, and transitions that result from an imperfect LLC. This is of particular importance, especially to handle LLC misses that can trigger evictions in the private cache (Problem 1). Unlike most of the existing works, we also experiment with a non-perfect LLC in Section V.

V. EVALUATION

Experimental Setup. We implemented PENDULUM* in an open-source cycle-accurate cache simulator which runs trace-based simulations [8]. We modelled a multi-core system consisting of four cores, c_0–c_3, with OoO pipeline. The private caches are 16kB with 64B cache line size, direct-mapped, and non-blocking. The LLC is an 8-way set associative cache which is perfect by capacity, unless otherwise stated. We define the hit latency, request latency, and data latency as 1, 4, and 50 cycles, respectively. We evaluate the system on benchmarks from SPLASH-2. Each core is mapped to one thread of the benchmark and are configured with timers obtained from the approach described in Section IV-C.

Predictability. In Figure 4, we compare WCML of all the cores in PENDULUM* with that of two counterpart solutions: PENDULUM representing the only real-time time-based solution, and State-of-the-Art (denoted as SoA), which represents predictable coherence with linear latency bounds [8]. The experimental WCML is always under the analytical WCML, demonstrating the predictability of PENDULUM*. By using timers to guarantee the hits, PENDULUM* achieves, on average, 2.15× reduction (Up to 4.9×) in WCML, compared to the SoA. Meanwhile, PENDULUM shows the worst WCML bounds. On average, PENDULUM is worse than PENDULUM* by 16×, and up to 36×, in terms of WCML. This is because PENDULUM's WCL^{miss} has a quadratic relation with N as opposed to the linear relation in SoA and PENDULUM*. Moreover, just like SoA, PENDULUM cannot guarantee any hits from isolation, so it assumes all memory accesses as *miss* further degrading its WCET analysis.

Average Performance. In this experiment, we compare all the three solutions against the conventional performance oriented COTS MSI. The average slowdown against the standard MSI protocol with a First-Come First-Serve (FCFS) arbiter in Figure 5 is 1.13× in PENDULUM*, 1.18x in SoA×, and 1.5× in PENDULUM. PENDULUM's degraded performance is mainly attributed to its TDM arbiter which suffers from latency due to idle slots.

Imperfect LLC. In this experiment, we consider an LLC with a finite capacity (128 kB) which has access to the main memory, modelled as a fixed latency DRAM with an access time of 100 cycles. The interference due to DRAM is an additive component to WCL^{miss}. For SoA, the additive component is $L^{arb} + L^{DRAM}$, where L^{arb} is the arbitration latency, and L^{DRAM} is the DRAM latency. This is because in worst-case, the data arrives at LLC just after the requesting core loses its access to the bus, so it must wait for an additional arbitration period. In PENDULUM*, however, the additive component is just L_{DRAM} because in the critical instance described in the proof of Lemma 1, only the first requesting core will suffer

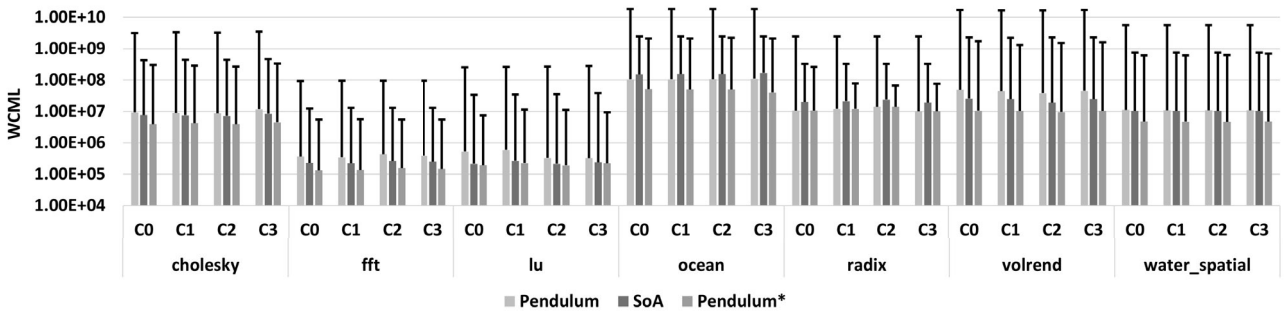

Fig. 4: Experimental (Solid bars) and analytical (T bars) WCML of all the tasks in a system of four critical cores running SPLASH 2 threads. The vertical axis is in logarithmic scale.

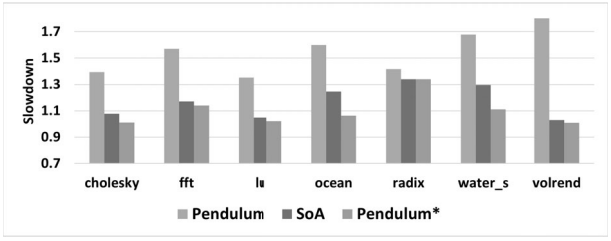

Fig. 5: Slowdown of the system consisting of all critical cores. Results are normalized against standard MSI execution time.

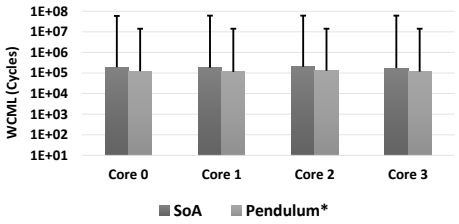

Fig. 6: Experimental (Solid bars) and analytical (T bars) WCML. y-axis is in logarithmic scale.

from the DRAM interference. The results of the experiment with fft benchmark are shown in Figure 6. Compared to SoA, PENDULUM* shows 4× improvement in the WCML bound.

VI. Conclusion

We identify two problems in multi-core real-time systems that void in-isolation cache analysis. Problem 1 occurs due to the presence of shared caches while Problem 2 occurs due to coherence interference. In this paper, we integrate time-based coherence with in-isolation analysis to preserve the hit/miss classification results from isolation. Using this method, we tighten WCML bounds by up to an order of magnitude in multi-core systems that allow predictable use of cache coherence with high performance.

References

[1] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A survey on static cache analysis for real-time systems," vol. 3, p. 05:1–05:48, Jun. 2016. [Online]. Available: https://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v003-i001-a005

[2] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, Nov. 2015.

[3] A. M. Kaushik, M. Hassan, and H. Patel, "Designing predictable cache coherence protocols for multi-core real-time systems," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2098–2111, 2021.

[4] A. M. Kaushik and H. Patel, "A systematic approach to achieving tight worst-case latency and high-performance under predictable cache coherence," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 105–117.

[5] N. Sritharan, A. Kaushik, M. Hassan, and H. Patel, "Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 433–445.

[6] M. Hassan, "Disco: Time-compositional cache coherence for multi-core real-time embedded systems," *IEEE Transactions on Computers*, pp. 1–14, 2022.

[7] S. Hessien and M. Hassan, "The best of all worlds: Improving predictability at the performance of conventional coherence with no protocol modifications," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020, pp. 218–230.

[8] M. Hossam and M. Hassan, "Predictably and Efficiently Integrating COTS Cache Coherence in Real-Time Systems," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Maggio, Ed., vol. 231. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 17:1–17:23. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2022/16334

[9] N. Sensfelder, J. Brunel, and C. Pagetti, "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 fuer Informatik, 2020.

[10] R. Pujol, H. Tabani, J. Abella, M. Hassan, and F. J. Cazorla, "Empirical evidence for mpsocs in critical systems: The case of nxp's t2080 cache coherence," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1162–1165.

[11] Y. Yao, W. Chen, T. Mitra, and Y. Xiang, "Tc-release++: An efficient timestamp-based coherence protocol for many-core architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 11, pp. 3313–3327, 2017.

[12] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache coherence for gpu architectures," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 578–590.

[13] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "Parallelism-Aware High-Performance Cache Coherence with Tight Latency Bounds," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Maggio, Ed., vol. 231. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 16:1–16:27. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2022/16333