Criticality and Requirement Aware Heterogeneous Coherence for Mixed Criticality Systems

Safin Bayes and Mohamed Hassan
McMaster University, Canada
{bayess,mohamed.hassan}@mcmaster.ca

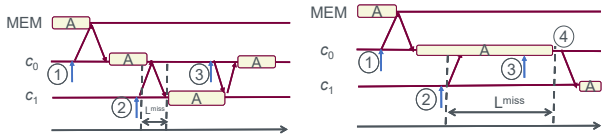
***Abstract*—We propose CoHoRT, as the first heterogeneous cache coherent solution for mixed criticality systems (MCS) equipped with several features that targets the characteristics and requirements of such systems. CoHoRT is requirement-aware. It provides an optimization engine to optimally configure the architecture based on system requirements. CoHoRT is also criticality-aware. It introduces a low-cost novel architecture to enable cores to heterogeneously run different coherence protocols (time-based and MSI-based protocols). Moreover, it enables a run-time switch between these protocols to provide hardware support for mode operation switch, which is a common challenge in MCS. Our evaluation shows that CoHoRT outperforms existing solutions both from worst-case memory latency as well as overall average performance. It also illustrates that CoHoRT is able to meet timing requirements in various MCS setups and showcases CoHoRT's ability to adapt to mode switches.**

I. Introduction

Modern embedded systems are Mixed Criticality Systems (MCS) executing tasks of different criticality levels, where a task's criticality is attributed to the failure consequences of this task. For example, in a self-driving car activating airbags is more critical than operating the infotainment system. Four main challenges with such systems inspire this work. **Challenge 1:** ***Heterogeneous Multiple Processor Systems-on-chip (MPSoCs)*** are increasingly deployed in MCS encompassing heterogeneous processing elements such as CPUs (real-time and non real-time), GPUs, DSPs, and dedicated accelerators. Tasks executing on these processing elements need to share data towards achieving the overall system's operations. Coherently sharing data in heterogeneous MPSoCs is a big challenge on its own [1], put aside supporting MCS. **Challenge 2:** Tasks with ***different criticalities*** share same hardware platform and contend on resources. **Challenge 3:** Tasks of same (or different) criticality still have their own ***distinct timing requirements*** based on what goal the task is accomplishing in the system. Some tasks are extremely timing sensitive; and thus, care the most about worst-case execution time (WCET) that should never exceed their designated deadlines. Other tasks can tolerate few deadline misses or might not even have tight timing requirements but instead require a certain average-case throughput. Guaranteeing all tasks same service (even if fairly done) is not a wise approach as it does not take into account these requirements; and therefore, is susceptible to either over-allocation or under-allocation of services/resources to tasks. **Challenge 4:** MCS can operate on ***different modes*** [2], which might be due to interacting with a dynamically-changing physical environment [3] or simply representing different failure modes [4]. Accordingly, the WCET of a task becomes a function of the operational mode (or the required level of assurance at that mode). In the original MCS model by Vestal, only two levels (and modes of operations) were considered, normal and critical [5]. This was later extended to support several criticality levels and modes of operation [6]. The DO-178C avionics standard defines five levels of assurance, whereas the ISO-26262 defines four automotive safety integrity levels.

Motivated by these challenges, we propose CoHoRT by making the following contributions. 1) CoHoRT presents the first heterogeneous cache coherence solution for MCS and real-time systems, where different processing elements are able to run different coherence protocols; either time- or snoop-based coherence while all the processing elements in the MPSoC remain fully coherent regardless of their protocol. While snoop-based protocols are commonly found on CPU clusters with small number of big cores, time-based coherence is more efficient for GPUs [7], [8] and for many tiny-cores CPUs [9] (**Challenge 1**). In this paper, we highlight the trade-offs of both protocol families and articulate why combining them in a heterogeneous solution best-fits MCS (Section III-A). 2) CoHoRT represents the first requirement-aware coherence solution for real-time systems (**Challenge 3**). The hardware offers configurable registers that dictates the amount of time a core will be able to maintain a cache line in its private cache regardless of the activity of other cores (Section III-B). We formalize an optimization problem that optimizes the timer values to meet system requirements while maximizing performance (Section V). 3) CoHoRT represents the first criticality-aware coherence solution that supports both any number of criticality-levels (**Challenge 2**) and criticality mode switches (**Challenge 4**). To enable mode switching (where the same task can generally have different requirements for each mode, CoHoRT introduces a novel low-cost architecture, where a core is able to switch between time-based coherence and snoop-based coherence by only configuring its timer-register and by adding a very small logic to the cache controller (Section III-B). We show that this switch enables better memory time budgeting when mode switches occur in MCS. Such solution represents one of the early approaches towards hardware/software codesign in MCS where hardware can cooperate with system scheduler to provide better system mode switching mechanisms, where less critical tasks might not be blindly forced to suspension (Section VI).

978-3-9826741-0-0/DATE25/© 2025 EDAA



(a) Snoop-based (b) Time-based

Fig. 1: Trade-offs between Snoop- and time-based coherence.

II. System Model

The considered system is a multi-core MCS of N cores running a set of mixed criticality tasks. A task τ_j, is characterized by the tuple, $< l_j, \Lambda_j, \Gamma_j^{m_l} >$, where l_j is the task criticality level, Λ_j is the total number of memory accesses, and $\Gamma_j^{m_l}$ is the WCML requirement of τ_j at mode m_l. Each core is associated with a criticality level, l_i, where $1 \leq l_i \leq L$. The system may operate on a set of modes $\mathcal{M} = \{m^l | 1 \leq m \leq L\}$. We do not impose constraints on how task scheduling or task-to-core mapping is done. At any time instance, the core inherits the criticality of the task running on the core in this instance.

Micro-architecture. Each core has a private L1 cache, and cores have have access to a shared memory which can either be a last-level cache (LLC), an off-chip DRAM, or both. The shared memory is inclusive of the L1 cache. The L1 caches and the shared memory exchange messages and data through a shared bus on which cores are granted access based on a predictable arbitration mechanism.

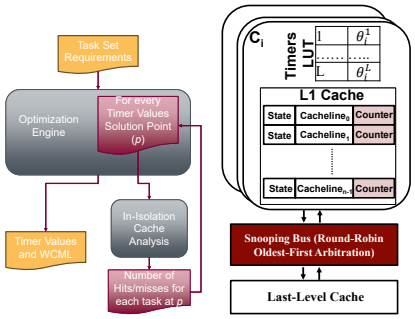
Mode Switching. A MCS operating at mode m^l might need to switch to a mode $m^{(l+1)}$ due to external (operating environment) or internal (failure) reasons. Upon such a switch, cores/tasks with $l_i < m+1$ operate at a degraded state. In our solution, this degraded state means they switch to MSI snooping based coherence.

III. Proposed Solution

The crux of our approach is to enable heterogeneous and adaptive coherence solution that is requirement- and cirticality-aware, and thus, meets the needs of modern MCS.

A. Motivation: Heterogeneous coherence for MCs

All existing predictable coherence solutions for real-time systems are not heterogeneous (i.e. support only a single protocol on the platform), with the majority being snoop-based [10]–[15], while few solutions explores time-based protocols [16], [17]. We start by explaining our rationale that combining these two families of protocols in a heterogeneous solution has the potential to better address MCS challenges. Under normal snoop-based coherence, a core has to give up the ownership of a cache line immediately once requested by another core based on coherence protocol specification. In contrast, a time-based protocol permits a core to keep the cache line valid in its private cache for the timer period; and hence, entertain hits to this line. Comparing these two behaviors, we make the following key observation. *Snoop-based protocols lead to a less worst-case miss latency for the requesting core, while it breaks timing-isolation among cores. On the other hand, a time-based protocol can restore the time-isolation at the expense of larger worst-case miss latency*. We use the example scenario in Figure 1 to elaborate. c_0 issues a store request to cache line A owned by the shared memory ①, memory sends the request to c_0's private cache and the latter is now the owner of A. Both snooping- and time-based protocols behave the same up until now. At ②, c_1 sends a store request to A. Under snoop protocols in Figure 1a, c_0 immediately invalidates its copy of A and gives up ownership to c_1. This leads to the short L^{miss}. However, by c_0 losing A due to interference from c_1, its subsequent requests to A that were supposed to be hits (under no interference) turn to misses. This is the case for request ③ in Figure 1a. Contrarily, under the time-based protocol in Figure 1b, c_0 upon receiving c_1's request at ②, it does not respond immediately; instead, it keeps ownership of A until timer expiration (④) and then hand it over to c_1. This protects c_0's subsequent hits to A during this period (thus, ③ is a hit), at the expense of the large L^{miss} latency compared to snoop protocols.



(a) Optimization (b) Architecture

Fig. 2: Proposed Solution.

We draw from this observation that in MCS, ideally a one wants to achieve these goals: 1) time-isolate high-criticality tasks from low-criticality tasks, or at least minimize and bound the impact of the later on the former (time-based is appealing here); 2) reduce the worst-case miss latency to meet task's timing-requirements (snooping is more appealing here and for time-based coherence, the smaller the timer value the less the miss latency is); 3) increase the overall system performance, especially for throughput oriented tasks running on them (time-based is more appealing here as it enables streaming hits to cache lines before giving them up; the larger the time period the better). Since these are clearly contradicting requirements, CoHoRT addresses them by proposing a heterogeneous solution combining both time- and snooping-based coherence, while formulating an optimization problem to find the system configuration that meets these requirements.

B. System Architecture

In this section, we introduce the architectural components, which are shown in Figure 2b.

Cache controller. In the cache controller, we introduce two new components. 1) One 16-bit counter per-cache line to monitor the lifetime of the cache line in the core. We find 16-bit for the registers and the counters to be sufficient. This adds only around 3% overhead for a 64B cache line. 2) A look-up table (LUT) which maps the timer values

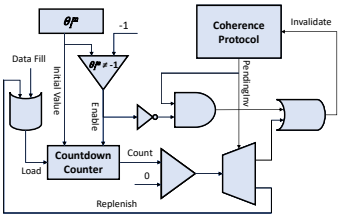


Fig. 3: High-level circuit design for heterogeneous coherence.

required for different modes of operation. Each row is indexed by the mode and comprises a 16-bit field for the timer of that mode. For 5 levels of criticality (e.g. in Avionics), this adds only a negligible 80 bits. Mode switching operation is discussed comprehensively in Section VI. For now, this section establishes the following terminologies and operations for the timers to support heterogeneous coherence. The timer threshold for a core c_i operating at mode m is defined by θ_i^m. Here, Θ is the the set of coherence timer registers and is denoted as $\Theta = \{\theta_i^m | \forall i \in [0, N-1] \ \forall m \in [1, L]\}$, where $\theta_i^m \in \{\mathbb{Z}^{0+} \cup \{-1\}\}$. $\mathbb{Z}^{0+}$ is the set of non-negative integers and θ_i^m represents the timer constant for c_i. $\theta_i^m = 1$ indicates that such core will not cache any line in its private cache, it will instead serve the pending request(s) and invalidate immediately. To model heterogeneous coherence, $\theta_i^m = -1$ is used as a special value which reduces the coherence protocol of c_i to the standard MSI protocol. Under this standard protocol, c_i can keep the cache line in its private cache until it has to be invalidated due to 1) a store request from another core, 2) back-invalidation from the LLC, or 3) its own replacement policy. CT^L is the set of coherence timer counters of cache line L in all critical cores and is denoted as $CT^L = \{ct_i^L | \forall i \in [0, N-1]\}$, where ct_i^L is an integer such that $-1 \leq ct_i^L \leq \theta_i^m$. Once a cache line is fetched to c_i's private cache, the counter will be set to the timer register value ($ct_i^L = \theta_i^m$). Then, the counter will be decremented by one in every cycle until it reaches 0. Once the counter reaches zero, the core will invalidate the cache line if there is a request from another core. Otherwise, it replenishes the counter by resetting its value to the timer threshold register. For the case of $\theta_i^m = -1$, the counters will not be decremented at all.

Shared Bus and Arbitration. We adopt the Round-Robin (RR) Oldest-First (RROF) arbitration mechanism [18].Similar to RR, RROF grants access to cores in a cyclic manner through a sequence. However, in RROF, a core does not lose its order in the sequence until its oldest request is served. This property allows us to obtain a tighter bound for the per-request latency as we will derive in the following sections.

A Low-Cost Heterogeneous Coherence. We propose a novel architecture design that enables a low-cost heterogeneous cache coherence, where cores can deploy one of two classes of coherence protocols: traditional snooping (MSI-based) protocols, or time-based coherence protocols. The basic intuition of this idea is to use a special value of the timer counter to determine whether the core has to give up the line's ownership immidately upon another core's request (modeling snoop-based) or keep it for the timer period (time-based). Figure 3 shows the high-level circuit design to support heterogeneous coherence with low overheads. The Countdown

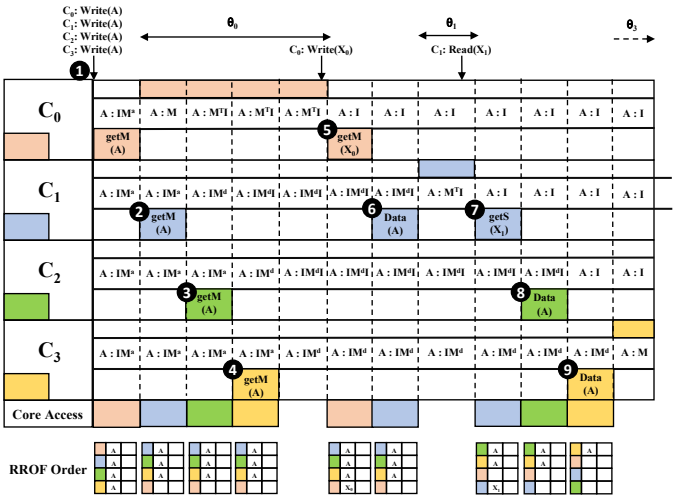


Fig. 4: Example operation of the proposed system architecture. c_0, c_1, and c_3 are critical. c_2 is non-critical. All cores initially issue a write request to cache line A.

Counter module takes the Load signal as input, which is triggered if the core receives a cache line or needs to replenish the counter. If Load is high, then the counter loads the initial value, θ_i^m, from the timer threshold register. The Enable signal is high if θ_i^m is not equal to -1. This signal indicates if the counter must decrement at each cycle. The output of the counter is the decremented value, Count. Count equals to zero means the timer has expired. If there is another core requesting for the cache line, then it must be invalidated or else, the timer should be replenished. The demultiplexer determines if the cache line must be invalidated or replenished with the help of the PendingInv signal, which indicates if another core is waiting for the cache line. This describes the general operation of time-based coherence. The time-based protocol can be reduced to the behavior of the standard MSI coherence using a special value for the timer, $\theta_i^m = -1$. A comparator checks if the timer threshold register stores the special value. If $\theta_i^m = -1$, then Enable is low, which indicates that the counter is disabled. If Enable is low and PendingInv is high, the core will invalidate the cache line, i.e, when $\theta_i^m = -1$, the core will not invalidate the cache line until another core requests for it. This is the operation of MSI protocol.

C. Example Operation

Figure 4 shows an illustrative example of a quad-core system operating under the proposed architecture, where c_0, c_1, and c_3 are running under the time-based protocol, while c_2 is running under the MSI protocol. 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 the figure. At ❶, as c_0 is at the top of the RROF order, it issues the request, receives the data, and starts its timer θ_0. Since c_0's oldest request is served at ❷, it loses its position in the RROF order. Also, c_1 issues its request at ❷, but does not lose its position in the RROF order because it has to wait for θ_0 to expire before its request can be served. Similarly, c_2 ❸ and c_3 ❹ issues their requests, and waits for the previous owner

to send the data. At ❺, 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 ❻ 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. Now, c_1 loses its position in the RROF order as its oldest request is served. Then ❼, θ_1 expires just when c_1 is issuing a request to X_1. At , as c_2 is at the top of the RROF order, it gets access to the bus and acquires the most updated copy of A from c_1 ❽. Since c_2 is running with MSI, it has to immediately give up the data to the next requester, c_3 ❾.

IV. Timing Analysis

Lemma 1: (**Per-Request WCL**.) The maximum WCL a request by a core, c_i, suffers can be calculated by Equation 1.

$$WCL_i^{miss} = SW + (N-1)\cdot SW + \sum_{\substack{j=0 \\ j\neq i}}^{N-1} \begin{cases} \theta_j + SW & \text{if } \theta_j \geq 0 \\ 0 & \text{if } \theta_j = -1 \end{cases} \quad (1)$$

Proof. The worst-case scenario for any request, $r_i(A)$ from a 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 (GetM(A)) one after the other. First, $r_i(A)$ must wait for the first core in the broadcast order to issue its request and receive the data from the shared cache. In worst-case, this consumes SW cycles, which is the first term in Equation 1. Cores operating under the time-based protocol will keep A for θ_j cycles. The timers may expire in the middle of a slot (1 cycle after the beginning of a slot in worst-case). So these cores must wait for *SW* cycles (in worst-case) before it can initiate data transfer to the next requester. These timer related latency components are represented by the third term in Equation 1. Finally, it takes an additional SW cycles to transfer the data from the owner core to the target core. This justifies the second term in Equation 1.

Definition 1: (**Worst Case Memory Latency**, WCML) is the worst-case total memory latency suffered by the whole task and is given by the following equation, where L^{hit} is the *hit* latency, WCL^{miss} is the maximum latency for a *miss* in the private cache, and M^{hit} and M^{miss} are the total number of private cache *hits* and *misses* occured in the task, respectively.

$$WCML^{timed} = M^{hit}\cdot L^{hit} + M^{miss}\cdot WCL^{miss} \quad (2)$$

M^{hit} and M^{miss} can be obtained from the in-isolation cache analysis by virtue of their timers [17]. However, for the tasks running on cores with the MSI protocol, it must be assumed all the requests are *misses* because the in-isolation analysis is not preserved without the timers. Therefore, Equation 2 reduces to the following for tasks running on cores with the MSI.

$$WCML^{snoop} = \Lambda \cdot WCL^{miss} \quad (3)$$

V. Obtaining Timer Values

To reiterate the trade-off subjected to Θ, increasing the timer allows the owner core to safeguard its cache line against evictions due to requests from other cores. This protection could potentially increase M^{hit} while decreasing M^{miss}. Conversely, this action also results in extended waiting times for other cores, as they must wait for the owner core's timer to expire, contributing to the increase of WCL^{miss} (Equation 1). In this section, we formalize an optimization problem to obtain the timer values.

Objective Function. The objective function minimizes the total average worst-case memory latency of the system which is expressed below.

$$\text{Minimize: } \sum_{i=0}^{N-1} \frac{M_i^{hit}\cdot L^{hit} + M_i^{miss}\cdot WCL_i^{miss}}{M_i^{total}}$$

Variables. 1) Set of timers (Θ): The main variable outcome of the optimization problem is the set of timers. Based on the solution point of the set of timers, the following parameters also are variables that are function of Θ. 2) The worst-case request miss latency per core (WCL^{miss}). 3) Hits and Misses per core (M^{hit}, M^{miss}). The lower bound of the timers is one cycle since it is the minimum value for which the core can be guaranteed with *hits*. The upper bound θ_{is} is the value of the timer for which the *hits* of c_i saturates. θ_{is} can be obtained by sweeping timer values for c_i in isolation: $1 \leq \theta_i \leq \theta_{is}, \forall i \ni \theta_i \neq -1$.

Constraints. All tasks running on a core with time-based coherence must satisfy their WCML requirements.

$$M_j^{hit}\cdot L^{hit} + M_j^{miss}\cdot WCL_j^{miss} \leq \Gamma_j, \forall j \ni \theta_i \neq -1 \quad \text{(C1)}$$

Solving. While the Θ-WCL^{miss} relationship can be described in a closed-form (Equations 1), capturing the Θ-M^{hit} relationship is not as straightforward as it is dependent on the application's memory behavior. As such, the objective function may not be differentiable with Θ. To address this problem, we integrate our cache static analysis model that we use to obtain the number of guaranteed hits for a certain θ with the optimization engine as follows. 1) The optimization algorithm (Genetic Algorithm (GA)) in our case) explores the solutions space to find the set of potentially optimal set of timers, Θ. 2) The optimization engine uses the cache analysis model as a black box to capture the Θ-M^{hit} by feeding it the found Θ and obtain the corresponding set of M^{hit}. 3) The engine then uses this set of M^{hit} to calculate the target function for this solution point. This flow is delinated in Figure 2a.

VI. Mode-Switching

MCS initially operate in a normal mode, $m_{l=1}$, where the memory requirements of all tasks are considered. Often times, these systems encounter unexpected scenarios where higher critical cores at risk of exceeding their WCET requirements. In such situations, the traditional approach is to switch the system to a different operational mode (usually referred to as degraded/critical mode), where the tasks of lower criticality levels are suspended. This approach is undesirable in industries because the lower criticality tasks are still crucial and suspending them can cause safety issues [19]. A previous work [20] has addressed this limitation of mode-switching at the interconnect arbitration level. Leveraging heterogeneous coherence, this section presents a mechanism which avoids

suspending tasks of lower criticality levels when switching between various modes of operation.

In our solution, When the system is operating on mode m_l, cores with a criticality level $l_i \geq l$ operate under time-based coherence, while cores with $l_i < l$ operate under normal MSI protocol. Thanks to our novel architectural approach, switching between time-based and MSI is performed by simply setting θ of that core as we discussed in detail in Section III-B. We use this mechansim as a hardware solution to mitigate the impact of mode switching on tasks with higher criticality, while only tune the service given to lower criticality without completely suspending them. Basically, by reducing the number of cores with time-based coherence, the higher criticality cores suffer from reduced timer-related interference (Equation 1). The side effect after reducing the coherence protocol to MSI for a core is that the number of *hits* are no longer guaranteed because the core is not guaranteed to keep the cache line after receiving the data. It is important to note that this still does not mean these cores do not have any guarantees, but instead it only loses the ability to reason about number of hits and hence all requests need to be assumed misses (which is the state-of-the-art bound). As a result, their WCML bound can be computed using Equation 3 assuming the remaining memory accesses as *misses*.

Now, we discuss how does the proposed solution switch between different modes. This is achieved by using the the Mode-Switch LUT in each core's private cache controller as shown in Figure 2b. The LUT stores the value of $\theta_i^{m_l}$ required for each mode. The LUT is indexed by the mode and stores the corresponding timer threshold in a 16-bit field. There are as many modes as the number of criticality levels. The hardware overhead for the LUT is minimal. For a system of five criticality levels, the Mode-Switch LUT requires an additional 80 bits. The LUT values are configured based on the offline process described in Figure 2a. First, the optimization engine (Section V) runs for every mode, m_l, takes all τ_j with $l_j \geq l$ as inputs with their requirements in that mode Γ_j^l and generates all θ_j^l, i.e, the timer values for mode m_l.

VII. Related Work

Among the cache coherence works in real-time systems [10]–[17], [21]–[23], PENDULUM [16] and CARP [13] are the only ones targeting multi-core MCS. PENDULUM introduced a predictable time-based protocol on top of the standard MSI protocol. Apart from the pessimistic WCET analysis, we identify additional limitations with PENDULUM. 1) It supports only two criticality levels; Cr and nCr. While the per-request access from Cr cores are bounded, non-critical cores have no guarantees. This is not suitable for acquiring certifications for modern MCS. For example, the DO178C certification used in avionics mandates guarantees for five criticality levels. 2) It employs an unfair arbitration scheme where nCr cores are granted access to the shared bus only if there are no requests from any Cr core. This grossly degrades the performance of nCr cores. CARP [13] supports the definition of several criticality levels; nonetheless, the solution handles all criticality levels (except the lowest) the exact same way.

Work Categories	Challenge 1 (Heterogeneity)	Challenge 2 (Criticality)	Challenge 3 (Requirements)	Challenge 4 (Mode Switching)
[10]–[12], [15], [21], [22], [24]	No	No	No	No
[13], [16]	No	Limited	No	No
[17]	No	No	Yes	No
CoHoRT	Yes	Yes	Optimized	Yes

TABLE I: Predictable Coherence Works and MCS challenges

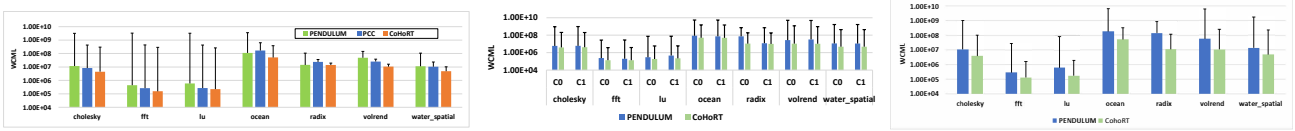
We find this to practically reduce the support to again only two levels where cores are either critical or non-critical. Similar to PENDULUM, CARP is also not requirement-aware, it gives all critical cores the same arbitration and coherence service regardless of their requirements. PENDULUM* [17] utilizes the timing based coherence to guarantee a number of hits for real-time cores; and hence, significantly improves the total task worst-case memory latency. While PENDULUM* achieves requirement-awareness, it does not support mixed criticality systems, nor mode switching. We summarize the limitations of existing works in addressing the MCS challenges in Table I.

VIII. Evaluation

Experimental Setup CoHoRT is implemented in Octopus, a cycle-accurate cache simulator [25]. The experiments are performed on a multi-core system consisting of four cores: c_0–c_3 with OoO pipeline. The cache hierarchy is composed of a private cache and a shared LLC. The private caches are 16kB with 64B cache line size, direct-mapped, and non-blocking allowing *hits-over-misses*. The LLC is an 8-way set associative cache. The hit latency, request latency, and data latency are set as 1, 4, and 50 cycles, respectively. The system is evaluated on benchmarks from SPLASH-2 [26], a multi-thread benchmark suite. Each thread of the benchmarks is mapped to one core. For obtaining timer values, we implement the optimization problem (Section V) in Matlab, where we use the GA with its default parameters. For the used SPLASH benchmarks in an core-i7 desktop, the optimization time ranges from 50 minutes (for *fft* with about 47k requests) to 20 hours (for *Ocean* with about 2.5M requests). Similar to the majority of existing coherence works, we show the results here for a perfect LLC to eliminate the interference from the off-chip main memory and focus on the overheads due to coherence interference. That said, we have also experimented with a non-perfect LLC including a fixed-latency main memory model. This experiment shows same observations as presented in this section and are hence omitted for conciseness [1].

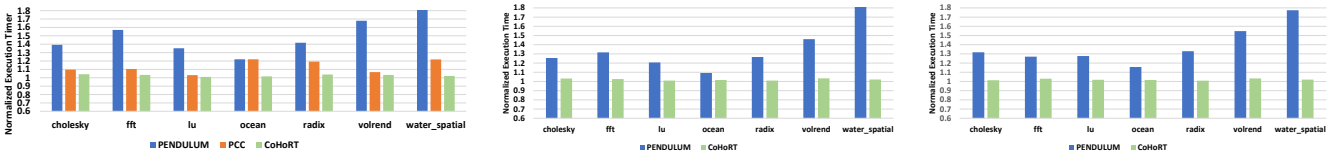
Total Worst Case Memory Latency. This section compares the total WCML of CoHoRT with PCC and PENDULUM under different configurations. **All Cr:** In this experiment, all four cores are configured as Cr. Figure 5a shows the comparison of WCML of all the cores in the different systems. The experimental WCMLs of CoHoRT are under their respective analytical WCML bounds, which shows that the solution is predictable. On average, CoHoRT is 2.15× tighter than PCC. Meanwhile, PENDULUM shows the worst WCML bounds. On average, PENDULUM is worse than CoHoRT by almost 16×. **2 Cr, 2 nCr:** In this configuration (Figure 5b), the WCML bounds of PENDULUM is worse than those of CoHoRT by approximately 6×. Since all the cores in the previous

[1]We intend to release the detailed implementation source code including both perfect and non-perfect LLC options for reproducibility upon acceptance.



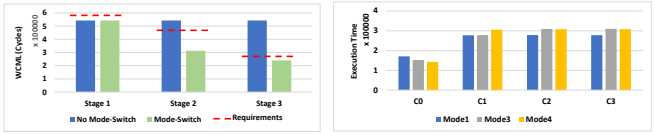
(a) All Cores are critical. (b) c_0, c_1: Cr. c_2, c_3: nCr. (c) Only c_0 is Cr.

Fig. 5: Experimental (Solid bars) and analytical (T bars) WCML. The vertical axis is in logarithmic scale.



(a) All Cr (b) 2 Cr, 2nCr (c) 1 Cr, 3nCr

Fig. 6: Overall system execution time in different configurations. Results are normalized against standard MSI.



(a) WCML of c_0. (b) Per-core performance.

Fig. 7: Mode-Switch Experiment.

configuration were Cr, each Cr was subjected to interference from three other Cr cores. But in this configuration, a Cr cores suffers from the interference of only one Cr core. **1 Cr, 3 nCr:** Figure 5c shows that on average, the WCML bounds of CoHoRT are 18× tighter. Results are significantly better than those of the prior configurations because the WCML improves considerably in CoHoRT while degrades severely in PENDULUM. In CoHoRT, cores do not suffer from the latency of its own timer. With no other co-runner Cr cores, the WCL effectively reduces to just the arbitration latency. Since the other cores have no real-time requirements, c_0 can have a very large timer to obtain as many *hits* without worrying about violating any constraints.

Average Case Performance. Figure 6 compares the performance of different systems in terms of the overall execution time normalized against the system executing with MSI protocol with a First-Come First-Serve (FCFS) COTS arbiter. In Figure 6a, the average slowdown is 1.03× in CoHoRT, 1.13× in PCC, and 1.50× in PENDULUM. The overall execution time is similar in the other configurations as well. Using the optimal timers allow the cores in CoHoRT to guard against interference from other cores without severely compromising the latency of the co-runner cores, resulting in better performance compared to PCC. The degraded performance in PENDULUM is mainly attributed to its TDM arbiter which suffers from unnecessary latency due to idle slots.

Criticality Levels and Mode-Switching This experiment investigates the capabilities of the adaptive mechanism proposed in Section VI. The MCS for this experiment consists of cores, c_0, c_1, c_2, and c_3 with criticality levels 4,3,2, and 1 respectively. The timer configurations of the cores at different modes, computed offline, is recorded in Table II. Figure 7a shows the requirements of c_0 changing at three different stages. Comparison of the system with and without mode-switching is shown. In this experiment, we use the *fft* benchmark from the SPLASH suite. In Figure 7a, at Stage 1 the system operates at mode-1 with all cores running under the time-based coherence. At Stage 2, the requirement of c_0 reduces by almost 1.5×. As shown in the figure, without mode-switch, the WCML bound of c_0 exceeds the requirement. As the situation calls for mode-switch, the system starts by adjusting c_3's coherence protocol to MSI and switch to mode-2. However, even at mode-2, the bound of c_0 exceeds the requirement, so the system switches to mode-3 where both c_2 and c_3's coherence has been reduced to MSI. Then, at Stage 3, the requirement of c_0 is further reduced by 1.8×, so the system switches to mode-4, where c_0 is the only core operating under time-based coherence. Thus, by reducing the coherence protocol of the lower critical cores to MSI, c_0 is able to satisfy its requirements as it gets tighter. Without mode-switching, the system would be unschedulable at Stage 2 and Stage 3.

TABLE II: Timer configurations of cores at different modes.

m	θ_0^m	θ_1^m	θ_2^m	θ_3^m	m	θ_0^m	θ_1^m	θ_2^m	θ_3^m
1	300	20	20	20	**3**	300	10	-1	-1
2	300	20	20	-1	**4**	500	-1	-1	-1

IX. Conclusion

CoHoRT proposes the first heterogeneous requirement- and criticality-aware cache coherence solution for MCS, where tasks of different criticality levels with varying requirements share the memory resources. Cores are able to exercise different coherence protocols; namely, time- or traditional snooping-based coherence based on their criticality and/or requirements. To achieve this, e proposed a novel low-cost approach to switch between the two protocols based on the configured value of the timer registers. We also formulated an optimization problem to configure the architecture based on the system requirements. Finally, we discussed how CoHoRT provides a hardware support to enable mode switches in MCS.

REFERENCES

[1] N. Oswald, V. Nagarajan, D. J. Sorin, V. Gavrielatos, T. Olausson, and R. Carr, "Heterogen: Automatic synthesis of heterogeneous cache coherence protocols," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 756–771.

[2] V. K. Sundar, S. Ramanathan, and A. Easwaran, "Design and analyses of functional mode changes for mixed-criticality systems," *Springer Real-Time Systems (RTS)*, 2023.

[3] H. S. Chwa, K. G. Shin, H. Baek, and J. Lee, "Physical-state-aware dynamic slack management for mixed-criticality systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

[4] A. Esper, G. Nelissen, V. Nélis, and E. Tovar, "An industrial view on the common academic understanding of mixed-criticality systems," *Springer Real-Time Systems (RTS)*, 2018.

[5] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *IEEE International Real-Time Systems Symposium (RTSS)*, 2007.

[6] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, pp. 1–69, 2013.

[7] 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.

[8] A. Tabbakh, X. Qian, and M. Annavaram, "G-tsc: Timestamp based coherence for gpus," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 403–415.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] A. M. Kaushik, P. Tegegn, Z. Wu, and H. Patel, "Carp: A data communication mechanism for multi-core mixed-criticality systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2019.

[14] 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.

[15] 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

[16] 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.

[17] S. Bayes, M. Hossam, and M. Hassan, "Shared data kills real-time cache analysis. how to resurrect it?" in *IEEE Design, Automation Test in Europe Conference Exhibition (DATE)*, 2024.

[18] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "Parallelism-Aware High-Performance Cache Coherence with Tight Latency Bounds," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2022.

[19] P. Graydon and I. Bate, "Safety assurance driven problem formulation for mixed-criticality scheduling," in *Proceedings of the Workshop on Mixed-Criticality Systems*, 2013, pp. 19–24.

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

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

[22] Z. Wu, M. Bekmyrza, N. Kapre, and H. Patel, "Ditty: Directory-based cache coherence for multicore safety-critical systems," in *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023.

[23] N. Sensfelder, J. Brunel, and C. Pagetti, "Modeling cache coherence to expose," in *ECRTS 2019*, 2019.

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

[25] M. Hossam, S. Hessien, and M. Hassan, "Octopus: a cycle-accurate cache system simulator," *IEEE Computer Architecture Letters*, 2024.

[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 24–36. [Online]. Available: https://doi.org/10.1145/223982.223990