CARP: A Data Communication Mechanism for Multi-Core Mixed-Criticality Systems

Anirudh Mohan Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren Patel
Department of Electrical and Computer Engineering
University of Waterloo
Ontario, Canada
{anirudh.m.kaushik, ptegegn, zhuanhao.wu, hiren.patel}@uwaterloo.ca

***Abstract*—We present CARP, a predictable and high-performance data communication mechanism for multi-core mixed-criticality systems (MCS). CARP is realized as a hardware cache coherence protocol that enables communication between critical and non-critical tasks while ensuring that non-critical tasks do not interfere with the safety requirements of critical tasks. The key novelty of CARP is that it is *criticality-aware*, and hence, handles communication patterns between critical and non-critical tasks appropriately. We derive the analytical worst-case latency bounds for requests using CARP and note that the observed per-request latencies are within the analytical worst-case latency bounds. We compare CARP against prior data communication mechanisms using synthetic and SPLASH-2 benchmarks. Our evaluation shows that CARP improves the average-case performance of MCS compared to prior data communication mechanisms, while maintaining the safety requirements of critical tasks.**

I. Introduction

Mixed-criticality systems (MCS) consist of tasks with varying safety requirements [1]–[4]. These tasks are typically categorized into different criticality levels based on the severity of consequences of any deviation in their safety requirements [5]. The avionic and automotive domains have adopted several principles behind MCS as seen in standards such as DO-178C, ISO-26262, and AUTOSAR [5]. These domains continue to notice an increase in demands for complex and integrated functionalities whose implementations require interactions and communication between these complex functionalities [6], [7]. There is considerable interest in the community in using *multi-core platforms* to deploy such functionalities [8]–[12] for resource consolidation and cost reductions.

There are several prior research efforts in deploying MCS on multi-cores, but many of them assume that tasks do not communicate with each other [13]–[20]. This assumption is *not representative* of practical systems. For example, Hamann et al. [7] described that communication is prevalent in automotive embedded applications deployed on multi-cores. Hence, it is not surprising that recent research efforts have attempted to support communication between tasks in MCS [6], [21]. We are encouraged by this trend to *explore predictable data communication mechanisms for MCS*.

We observe that prior efforts on designing data communication mechanisms for MCS have the following side-effects: (1) they *underutilize* the performance opportunities available on multi-cores, and (2) they *disallow* communication between critical and non-critical tasks [6]. For the first side-effect, consider Chisolm et al.'s [6] work, which allows communication between critical tasks, but these tasks must be executed on the *same* core of a multi-core platform. Using this approach, if all tasks were to communicate with a common task, then they would all have to be deployed on the same core. This would result in underutilization of the multi-core platform (effectively single-core utilization). We find that this side-effect is a result of *certain guidelines put forth by the standards or lack thereof*. Therefore, any data communication mechanism that complies with the standards may also suffer from the same side-effect. Consider the AUTOSAR standard that allows data communication between critical tasks as long as they reside in the same memory partition (§ 2.1.2.2 and § 2.1.2.4) [22]. However, AUTOSAR also mandates that tasks sharing a memory partition must be executed on the same core (§ 2.7) [23], which restricts the utilization of hardware parallelism offered by multi-cores.

For the second side-effect, to the best of our knowledge, there is limited guidance in the standards on the requirements of communication between critical and non-critical tasks. Consequently, the general approach taken has been to disallow communication between such tasks. However, we find that practical deployments can benefit from communication between non-critical and critical tasks. Examples include the use of non-critical tasks for run-time monitoring where tasks monitor the execution of critical tasks for data correctness [24]–[26], and quality management tasks [27] that improve the overall functioning and responsiveness of the MCS. Presently, one way to incorporate such non-critical tasks in MCS is to *elevate* them to be critical tasks, which would most likely impact the safety-critical requirements of critical tasks. Note that ISO-26262 allows non-critical tasks to co-exist with critical tasks in the same memory partition as long as the safety requirements of critical tasks are not violated (§ 2.7) [27]. To collect on the potential benefits of incorporating non-critical tasks, communication must be allowed with critical tasks, but it must be enabled carefully to ensure predictability, and without affecting the safety-critical requirements of critical tasks.

In this work, we develop a data communication mechanism for MCS that (1) leverages performance opportunities offered by multi-core platforms, and (2) allows communication between critical and non-critical tasks without violating the safety-critical requirements of critical tasks. In particular, we focus on the safety requirements that deal with the *temporal properties* (worst-case latency bounds) of critical tasks. To accomplish this, we cautiously and excitedly step beyond the constraints placed by the standards with the hope of fostering discussions on proposed extensions for future evolution of the standards. To motivate benefits of such proposed extensions, we design CARP, a data communication mechanism based on hardware cache coherence that uses these proposed extensions to enable communication between critical and non-critical tasks, and deploys them across multiple cores. A key novelty of CARP is that it *dynamically* handles the communication between tasks based on their criticality levels. As a result, CARP is a *criticality-aware* data communication mechanism. This criticality awareness property allows critical and non-critical tasks to communicate such that the temporal properties of critical tasks are not affected by non-critical tasks. Prior works that facilitate data communication between critical and non-critical tasks such as [28] do not dynamically adapt the communication, and hence, introduce some timing interference in the temporal bounds of critical tasks. From our evaluation, CARP improves average-case performance by 30% over prior state-of-the-art data communication techniques proposed for MCS and real-time systems.

Our **main contributions** in this work are as follows.

- We propose CARP, a criticality-aware cache coherence protocol that enables high performance data communication between critical and non-critical tasks while ensuring the non-critical tasks do not interfere with the worst-case latency bounds (WCL) of critical tasks.
- We present a latency analysis for CARP to derive the WCL bounds on data communication.
- We compare CARP against prior approaches for enabling data communication using synthetic and SPLASH-2 workloads [29]. We show that the observed data communication latencies are within the WCL bounds, and CARP offers improved average-case performance over prior approaches for predictable data communication.
- We open-source the implementation of CARP at https://git.uwaterloo.ca/caesr-pub/mcs-carp to allow the research community to explore and build new data communication mechanisms for MCS using hardware cache coherence.

II. System Model

We denote a task set with $\mathbb{T}$ tasks in the system as $\Gamma = \{{\tau_i}^l : l \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}, i \in [0, \mathbb{T}-1]\}$ where every task has a criticality level l [5]. Our MCS model follows standards in avionics and automotive domains that classify tasks into different criticality levels based on their safety requirements [22], [27], [30], [31]. A task may have one of the five criticality levels: level A tasks are the most critical whose failure may result in fatalities through level E that are not critical and experience system performance impacts on a failure. We collectively refer level A-D tasks as *critical tasks* and level E tasks as *non-critical tasks*. Levels A and B tasks mandate tight WCL bounds with level A tasks having more stringent requirements than B. Tasks at levels C and D are soft real-time tasks that also need WCL bounds; however, these bounds are less stringent than levels A and B tasks. Level E tasks do not have any WCL bounds. Such a MCS model has been used in prior research [6], [17], [20], [32] [1].

Our real-time multi-core platform has N cores $C = \{c_0, c_1, ..., c_{N-1}\}$. A task mapped onto a core inherits the task's criticality level. For example, ${\tau_i}^l$ mapped onto core c_j results in c_j^l indicating that a task of l criticality level is executing on c_j. Note that we do *not* constrain a core to run a single task, and multiple tasks with different criticality levels can be executed on the same core. We require the core to identify the criticality level of the task currently executing on it. We describe one architectural extension to achieve this identification in Section VI. For the remainder of the text, we use the term core to refer to the task executing on the core. We denote $C^l = \{c_i^m : \forall c_i^m, m = l\}$ as the set of cores running tasks of criticality l. We assume that cores are in-order and allow for at most one outstanding memory request. Our evaluation empirically validates the analysis with this assumption. However, the proposed solution is independent of the core architecture, and works with out-of-order cores. The cores have a private memory hierarchy with caches and shared memory. The caches hold a subset of data stored in the shared memory, and the shared memory holds all the data needed by tasks running on the multi-core platform.

Cores communicate with the shared memory through a shared bus as the interconnect. A shared bus deploys an *arbitration policy* that manages the communication over the bus. The arbitration policies place constraints on when cores and memories are granted access to the shared bus for communication, and/or the amount of bus bandwidth made available for cores and memories. We deploy our proposed data communication mechanism on variants of time-division multiplexing (TDM) and round-robin (RR) arbitration policies. These arbitration policies have received considerable attention in the real-time community [14], [15], [33]–[36], and have been implemented in real-time platforms [37], [38].

III. Motivation

We list two key guidelines defined in the AUTOSAR standard that govern the design of data communication mechanisms. The *first* guideline allows tasks of different criticality levels to reside in the same memory partition (§ 2.1.2.2 and § 2.1.2.4) [22]. A consequence of this guideline is that tasks can communicate through shared data [6], [7]. Therefore, tasks of any criticality level can communicate with each other through *shared data* resident on the same memory partition. The *second* guideline states that tasks sharing a memory partition must execute on the same core (§ 2.7) [23]. This guideline forces tasks communicating via shared data to reside on the same core. There are two key limitations that these guidelines

[1]We are aware that ISO-26262 and AUTOSAR standards define level D as the highest criticality level and level A as the lowest criticality level.

TABLE I: AUTOSAR guidelines satisfied and extended by CARP.

CARP feature	Standard guideline	Relationship
Multiple tasks share a memory partition	§ 2.1.2 [22]	Satisfies
Tasks of different criticality levels share a memory partition	§ 2.1.2 [22]	Satisfies
Critical and non-critical tasks share a memory partition	§ 2.7 [27]	Satisfies
Data communication between non-critical and critical tasks	None	**Extends**
Tasks sharing a memory partition are deployed across cores	§ 2.7 [23]	**Extends**

impose for MCS deployments on modern and future multi-core platforms. (1) Limiting the number of cores that can be used based on data communication patterns of the application. As multi-cores continue to have large core counts, there would be considerable underutilization of hardware resources in deployments where tasks communicate. (2) Deploying tasks onto processing elements best suited for their execution is limited. Heterogeneous multi-core platforms with various accelerators [12] match the needs of modern applications, and has received recent attention for MCS [39], [40]. For example, certain machine-learning or vehicle tracking functionality may use a graphics-processing unit, and other computations may use real-time cores. However, if these functionalities communicate, then such heterogeneous platforms cannot be used since the tasks must reside on the same core.

In an effort to address these limitations, we present two possible extensions for consideration. The first *extension* allows communicating tasks to be *deployed* across multiple cores provided the safety-critical requirements of critical tasks are not violated. Recently, Hassan et al. [41] proposed an approach to allow communicating tasks of the same criticality level to execute on different cores. They showed significant performance improvements over deploying communicating tasks onto the same core while preserving safety-critical requirements. Our work distinguishes itself from [41] in that we focus on data communication across tasks of different criticality levels deployed across multiple cores.

The second *extension* allows communication between critical and non-critical tasks such that non-critical tasks do not violate the safety-critical requirements of critical tasks. To the best of our efforts, we did not find any guidance in AUTOSAR for data communication between non-critical and critical tasks. Hence, the only way to allow such tasks to communicate is to elevate the criticality level of non-critical tasks. However, the introduction of newly elevated critical tasks will interfere with the temporal requirements of existing critical tasks. Hence, an alternative mechanism that allows for such communication without impacting the temporal requirements of critical tasks is desirable. Note that ISO-26262 does allow non-critical tasks to co-exist with critical tasks in the same memory partition as long as non-critical cores do not violate the safety requirements of critical tasks (§ 2.7) [27], but the general approach to such communication has been to disallow it. In this work, we *disallow* level E tasks to potentially corrupt memory contents by restricting them to only *read* communicated data.

Table I summarizes the relationship between CARP's features and the guidelines described in the AUTOSAR and ISO-26262 standards. This relationship falls into 2 categories: (1) *Satisfies*: CARP's features satisfy the guidelines in the standards and (2) *Extends*: CARP's features require extensions to the standards. We view CARP as a step towards identifying the benefits of extending the guidelines on data communication in order to develop high performance yet predictable data communication mechanisms for multi-core MCS.

IV. Background

A. Hardware cache coherence

Hardware cache coherence is a hardware mechanism that provides all cores in a multi-core platform access to data that may be cached in their private caches [42]. A cache coherence protocol prescribes a set of rules to maintain a coherent view of data by ensuring that a core reads the most up-to-date version of the requested data. It further allows multiple cores to simultaneously have copies of data with the up-to-date value in their private caches. Each cache controller (CC) implements the cache coherence protocol, which consists of states that represent read and write permissions of data, and transitions between states that are triggered based on the cores' data communication activity. Based on the number of cores in the multi-core platform, communication between cores may occur over a snoopy shared bus or a NoC [42]. For this work, we limit our discussion to communication using a snoopy shared bus implementation that is appropriate for multi-core platforms with eight cores or less [8], [9], [11], [43].

Data communication begins when a core *generates* a memory request (read/write). This memory request first looks up the address in the core's private cache(s). If the address is found (cache hit), the private cache(s) returns the corresponding data, and the data communication is marked as completed. If the address is not found (cache miss), the cache controller *issues* a coherence message based on the type of memory access (read/write), and *broadcasts* on the snooping bus to all cores and the shared memory connected to the snooping bus. A coherence message is said to be *ordered* on the bus when all cores and the shared memory observe the broadcasted message on the bus. Data responses for data communication on the same data by different cores are returned based on the broadcasted order. The granularity of the data transferred is of cache line size, which is the unit of data transfer in the memory hierarchy.

A cache coherence protocol has *stable* and *transient* states. Stable states denote read or write permissions of a cache line, and transient states are intermediate states traversed while transitioning between stable states. Figure 1 shows the stable states of the **M**odified-**S**hared-**I**nvalid (MSI) cache coherence protocol, which is a base protocol upon which modern coherence protocols such as MESIF and MOESI [42] are built. Note that Figure 1 does not show the transient states and its transitions for brevity. A cache line in *Invalid* (*I*) indicates invalid data that cannot be used. A cache line in *Modified*

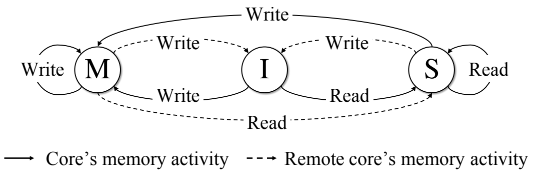


Fig. 1: Modified-Shared-Invalid (MSI) coherence protocol.

(M) indicates that the core modified the data in the cache line; hence, the core that made the modification *owns* the cache line, and has the most up-to-date value of the data. We refer to cores that own different cache lines as *owners*, and requesting cores that are not owners of a cache line as *remote cores*. There can be only one owner for a cache line. A cache line in *Shared* (S) indicates a cache line that was read, but not modified. Multiple cores may have the same cache line in the S state. This allows read hits in their respective private caches. This constraint of one owner for a cache line or multiple cores sharing a cache line is referred to as the single writer multiple reader (SWMR) invariant [42]. A core that has a private copy of data in M state must *write-back* the updated value to the shared memory on observing remote cores' memory activity on the same data.

Transient states are necessary to allow multiple cores to interleave their requests on the snoopy bus. Consequently, the snoopy bus implementation is non-atomic, and the primary reason for this is to offer improved performance [42], [44]. For such bus implementations, transient states capture state change activities from other cores to the same shared data. A core's request can experience intervening requests when waiting for its coherence message to be ordered on the bus, or when the core is waiting for the data after its request was ordered on the bus. Transitions between states are triggered based on the coherence messages broadcasted on the snooping bus. These coherence messages vary based on the memory requests generated by a core.

B. Predictable hardware cache coherence

Hassan et al. [41] proposed PMSI that provided the performance benefits of hardware cache coherence protocols while delivering predictability for multi-core real-time systems [41]. Our data communication mechanism is built using cache coherence, and utilizes the design guidelines and architectural extensions proposed in [41], which we briefly describe.

There are two architectural extensions that work with the PMSI cache coherence protocol. The first architectural extension is the pending request lookup table (PR LUT) at the shared memory. The PR LUT records pending requests to cache lines. Multiple pending requests to the same cache line are maintained in the PR LUT in broadcasted order. Data responses from the shared memory to pending requests to the same cache line are sent in broadcasted order. As a result, the PR LUT ensures predictable data accesses from the shared memory. The second architectural extension is in each core's private cache controller that comprises of two FIFO buffers: pending request (PR) buffer, and the pending write-back (PWB) buffer. The PR buffer holds requests for coherence

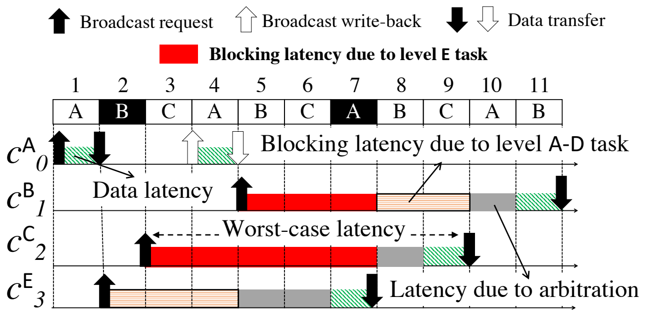


Fig. 2: Blocking communication due to shared memory responses.

messages that are ready to be broadcasted, and the PWB buffer holds pending write-back responses due to memory activity of other cores. When an owner observes a remote request, it marks the modified cache line for write-back and records the address in its PWB buffer. Write-back responses and requests access the shared snooping bus. This is because servicing requests in the PR buffer involves either broadcasting the request or receiving the requested data from the shared memory, and servicing write-back responses involves sending the write-back data to shared memory. Hence, each core deploys work conserving round-robin (RR) arbitration between servicing pending requests and write-back responses in the PR and PWB buffers to limit interference on the shared bus.

V. High level overview of CARP

CARP delivers performance benefits by allowing (1) communicating tasks to be distributed across cores, and (2) communicated data to be cached in the private caches of cores. It follows the template of guidelines put forth by Hassan et al. [41], and includes new features specifically catered for multi-core MCS. There are two interference scenarios that arise due to communication to/from level E tasks on the WCL bounds of critical tasks: (1) interference scenario due to data responses from shared memory and (2) interference scenario due to write-backs. CARP disallows these interference scenarios through *two* techniques: (1) forces level E cores to *abort-and-retry* in the presence of simultaneous communication from critical cores on the same shared data and (2) *partitions* the PWB buffer to isolate write-back responses for critical cores from non-critical cores, and schedule write-back responses from non-critical cores in slack. In the following section, we illustrate these interference scenarios and the impact of the techniques using a 4-core system ($c_0 - c_3$) that executes a level A, B, C and E task respectively. For ease of explanation, we consider a schedule that uses TDM arbitration. This schedule allocates one slot for each levels A-C cores to manage concurrent accesses to the shared memory. Following the guidelines set by Mollison et al. [32], we use slack to schedule data communication to/from E tasks.

A. Interference due to data responses from shared memory

Observation. Figure 2 highlights the interference scenario due to data responses from shared memory. In the first slot, c^A_0 broadcasts a write request to data X. The shared memory sends

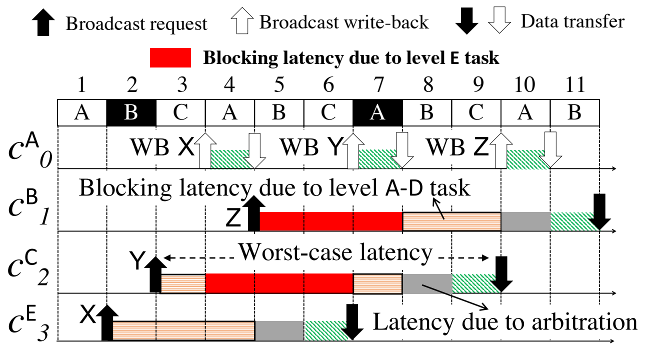


Fig. 3: Blocking communication due to write-back responses.

X in the same slot, and c_0^A completes its request. An updated copy of X resides in the private memory of c_0^A, and a stale copy of X resides in shared memory. The second slot, which is allocated to c_1^B, is unused by c_1^B making it a slack slot. Suppose c_3^E uses this slack slot and broadcasts a read request to X. Since, c_0^A has an updated version of X, it has to write back the update to the shared memory before the shared memory can send X to c_3^E. Hence, c_0^A must wait for its allocated slot to update the shared memory (slot 4). In the third slot, c_2^C broadcasts a write request to X. In slot 5, c_1^B issues a write request to X, and will receive X from the shared memory after c_2^C. After c_0^A updates the shared memory with the updated X, the shared memory can send data to the cores waiting on X. To maintain *data correctness*, the *shared memory must send data in the order of requests* [41], [42]. For example, if the shared memory reorders data response to c_1^B's request over that of c_2^C's request then c_1^B will receive a value X updated only by c_0^A and not by c_0^A and c_2^C. As a result, c_1^B operates on incorrect X value, which compromises data correctness. Hence, the shared memory must first send data to c_3^E and then to c_2^C and c_1^B. This *blocking* of data communication from the shared memory to c_2^C and c_1^B due to c_3^E is highlighted in red in Figure 2.

Solution. One potential solution is to *prioritize* data responses from shared memory to critical tasks over data responses to level E tasks. However, we observe that prioritization can *indefinitely* defer the data responses to level E tasks, which limits level E tasks' effectiveness to MCS functioning. For the example in Figure 2, consider that the shared memory prioritizes responses to c_2^C and c_1^B over the response to c_3^E. Since c_1^B does a store operation, the memory and c_1^B move to the modified state (M) after sending and receiving X respectively. After c_1^B completes the store operation, two conflicting scenarios exist that prevent c_3^E to receive X: (1) the shared memory cannot send X to c_3^E as it must wait for c_1^B to write-back the updated X, and (2) c_1^B does not mark X for write-back as it does not observe the pending read from c_3^E, which was issued *earlier* than c_1^B's request. Hence, an alternative solution that does not indefinitely defer data responses to level E tasks is necessary.

Revisiting the above example, we observe that if c_3^E *aborts* its current request to X on observing remote requests to X from critical cores and *retries* its request to X after observing requests from c_2^C and c_1^B, then c_1^B observes the request from c_3^E and schedules the write-back response for X. The order of requests observed by the shared memory will be c_2^C, c_1^B, and c_3^E, and the shared memory can send the updated X to c_3^E after c_1^B completes its write-back. Hence, the *abort-and-retry* mechanism ensures that critical cores will *not* be blocked by data responses to level E cores and level E cores will receive their data responses. Note that this mechanism in CARP offers a *trade-off* between the value received by a level E core for a request to shared data and the freedom from blocking due to communication to/from level E cores on the WCL bounds of critical cores. In particular, a level E core may receive a more updated value of the requested data compared to the value of the data when the level E core broadcasted its first request to the requested data. We find this trade-off to be acceptable as ensuring no interference from non-critical cores to the temporal requirements of critical cores is a key safety requirements in MCS [27], [45].

B. Interference due to write-back responses

Observation. Hassan et al. [41] proposed the PWB data structure in the CC to *isolate* requests made by a core that miss in the private cache and write-back responses due to memory activity from other cores [41]. They applied a predictable work conserving round-robin (RR) arbitration mechanism between the PR and PWB buffers as both cache miss requests and write-back responses require access to the shared bus [41]. However, their cache coherence protocol does not consider communication between mixed-critical and non-critical tasks [41]. Using Figure 3, we show that the current PWB design can cause blocking interference to critical cores in the presence of data communication between critical and non-critical cores.

Figure 3 modifies the example in Figure 2 such that c_3^E, c_2^C and c_1^B access different data blocks (X, Y, and Z respectively) that are modified by c_0^A and reside in c_0^A's private cache. Hence, c_3^E requests X in slot 2, c_2^C requests Y in slot 3, and c_1^B requests Z in slot 5. On observing these requests, c_0^A marks blocks X, Y, and Z for write-back by placing these blocks in its PWB buffer. Based on the guidelines listed in [41], write-back responses in a core's PWB are scheduled in a first-in-first-out (FIFO) order. Hence, in the first available slot of c_0^A that is marked for write-back, which is slot 4 in Figure 3, c_0^A will write-back X followed by write-backs to Y and Z in slots 7 and 10, respectively. Although the data requested by the critical cores are different from that requested by c_3^E, the FIFO order of draining the write-backs results in *blocking* communication to the critical cores as highlighted in Figure 3. In Figure 3, data response to c_2^C (Y) is blocked by the write-back response for c_3^E on X, which further blocks the data response to c_1^B (Z).

Solution. To eliminate this blocking interference, we *partition* the PWB of each core to isolate write-back responses for critical cores and non-critical cores, and schedule write-back responses from the partition containing write-back responses to non-critical cores in slack. Applying this approach for the example in Figure 3, c_0^A will schedule the write-backs to Y and Z in slots 4 and 7 respectively, and does not incur any blocking interference from c_3^E's write-back response.

The rationale for using slack to schedule write-back responses for non-critical cores is based on the observation that implementing a cache coherence protocol for data communication *increases* the availability of slack in the system. This occurs because cores experience a larger number of hits in their private caches, which reduces the number of accesses to the shared memory. In turn, this reduces the utilization of the cores' allocated slots rendering them to be slack. Our evaluation shows that when using cache coherence, up to 40% and 76% of the allocated slots are unused for synthetic and real-world benchmarks rendering them as slack. Conventional slack allocation policies allocated slack to low criticality and non-critical cores that have pending requests [32], [33], [46]. In this work, we propose a different slack allocation policy that allocates slack for ready requests from low criticality and non-critical cores *and* pending write-back responses from non-critical PWBs across cores. Scheduling write-back responses due to non-critical cores in slack ensures no blocking interference due to write-backs on the WCL bounds of critical cores.

Note that CARP allows *bounded* timing interference on the timing guarantees of levels A-D cores due to other levels A-D cores. This bounded timing interference comes from the (1) predictable arbitration on the shared memory and (2) read-write memory activity of other levels A-D cores. Timing interference from (1) is a natural consequence when arbitrating accesses to a shared resource. The ISO-26262 standard suggests that tasks of different criticality levels can co-exist in the same memory partition as long as a lower critical task does not interfere with the timing requirements of a higher critical task (ISO-26262-9 §6.5) [27]. Given the examples of timing interference listed in ISO-26262 (Annex D in ISO-26262-6) [27], CARP does not allow for unbounded blocking of execution.

VI. CARP IMPLEMENTATION

In this section, we describe implementations of the techniques presented in Section V. To implement abort-and-retry, level E cores must *differentiate* the criticality levels of requests broadcasted on the bus. Similarly, the PWB partitioning mechanism requires critical cores to *differentiate* between critical and non-critical requests to enqueue write-back responses in the appropriate PWB partitions. To this end, CARP's coherence protocol and architectural extensions enable CARP to be *criticality aware*. Figure 4 shows CARP's protocol state machine and the architectural extensions necessary to support CARP. CARP implements two coherence protocols: (1) for level A-D cores (Figure 4a) and (2) for level E cores (Figure 4b). Figure 4c shows the architectural extensions. Table II tabulates the transient states most relevant to the main contributions in this work, the events leading to the transient state (Cause field), and the operations executed by the CC in the transient state (Action field). A complete description of the protocol can be found in https://git.uwaterloo.ca/caesr-pub/mcs-carp.

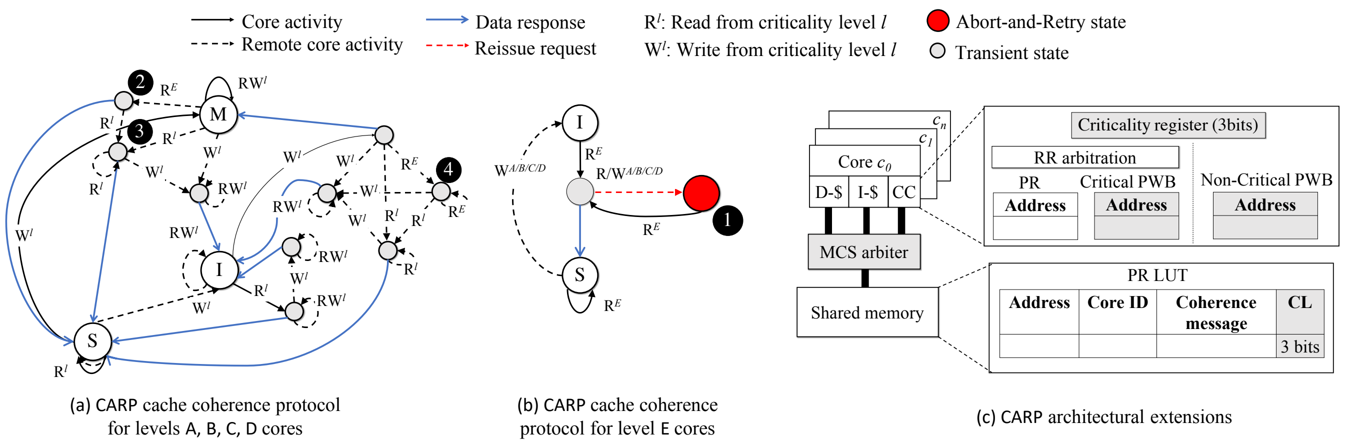
A. Implementing abort-and-retry for level E cores

CARP introduces a *criticality register* in each core's cache controller that tracks the criticality level of the current task scheduled for execution by the real-time operating system (RTOS) as shown in Figure 4c. This allows the cache controller to broadcast the criticality level. Initializing the contents of the criticality register can be done either by the RTOS scheduler, or by the task through software extensions prior to a task's execution. The cache controller (CC) looks up the criticality level in this register when generating coherence messages.

Figures 4a and 4b show the protocol specifications to support *abort-and-retry* memory requests. For each request generated by a core, the core's CC looks up the contents of the criticality register and broadcasts the request along with the core's criticality information. For critical cores, the CCs broadcast read (R) and write requests (W) as R^l and W^l respectively where $l \in \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}$ based on the criticality register contents. For non-critical cores, the CCs broadcast read requests as R^{E}.

Figure 4a shows CARP's protocol specifications for data communication between critical tasks in which all critical cores follow the *same* transitions and state changes for data communication between critical cores. On the other hand, Figure 4b shows CARP's protocol specifications for data communication between level E cores and critical cores in which different transitions are exercised based on the criticality levels of the remote requests. We revisit the example in Section V-A to highlight the state transitions for c_3^{E} in Figure 4b and for cores c_2^{C} and c_1^{B} in Figure 4a. c_3^{E} broadcasts its request to X, and transitions from I to a transient state that denotes c_3^{E} is waiting on data from the shared memory. The shared memory records c_3^{E}'s pending request in the PR LUT. c_0^{A} observes the level E remote read request on the shared bus, and updates its non-critical PWB with a pending write-back response to X. c_0^{A} then transitions from $M \rightarrow$ ❷. While c_3^{E} is waiting for the data response, c_2^{C} broadcasts a request to X. Due to this broadcast, two state transitions are exercised: (1) c_0^{A} observes a critical read request and transitions from ❷ $\rightarrow$ ❸ (Figure 4a) and updates its critical PWB with a pending write-back response to X and (2) c_3^{E} observes a critical read request and moves to the transient state denoted as ❶ in Figure 4b. At transient state ❶, c_3^{E} *aborts* its current request, and *retries* the read request to X by regenerating the read request.

Aborting a request requires discarding *all* information about the request from the system. In c_0^{A}, there are two pending write-back responses to X: (1) in the non-critical PWB due to c_3^{E}'s request and (2) in the critical PWB due to c_2^{C}'s request. At the PR LUT, there is an entry corresponding to requests from c_3^{E} and c_2^{C}. Hence, the aborting mechanism discards the write-back response in c_0^{A}'s non-critical PWB due to c_3^{E}'s request and the PR LUT entry corresponding to c_3^{E}'s request. Discarding entries in the non-critical PWB is done during the transition between transient states ❷ to ❸. Prior to updating the critical



(a) CARP cache coherence protocol for levels A, B, C, D cores

(b) CARP cache coherence protocol for level E cores

(c) CARP architectural extensions

Fig. 4: CARP protocol specification.

TABLE II: Transient states and transitions introduced in CARP.

Transient state	Cause	Action
AR (❶)	A level E core waiting for data response observes a remote read/write request from a critical core.	Level E core aborts and retries its request.
$MS^{wb}-$E (❷)	A critical core that has modified data in private cache observes a remote read request from level E core.	The critical core enqueues write-back response in non-critical PWB.
MS^{wb} (❸)	A critical core that has modified data in private cache observes a remote read request from another critical core.	The critical core enqueues write-back response in critical PWB and removes any matching entry in non-critical PWB.
IM^DS-E (❹)	A critical core waiting for data response to complete store operation (M) observes remote load request from level E core.	After completing store on data response, core enqueues write-back response in non-critical PWB.

PWB, the CC scans the non-critical PWB for write-back responses to the same data, and discards them. CARP extends the PR LUT with criticality information and introduces logic in the shared memory controller that discards entries for level E cores in the PR LUT when the shared memory observes critical cores' data communication on the same data.

The above set of transitions for c_3^{E} repeat on observing c_1^B's remote request to X. Hence, the final order of requests observed by the cores and shared memory to X is: c_2^{C}, c_1^B and c_3^{E}. If there are no other requests to X from other critical cores between c_3^{E} request and its response from shared memory, X in c_3^{E} transitions to S on receiving X from the shared memory. Note that in the presence of critical requests to the same shared data, E-cores can continuously abort and retry their memory requests. We allow for this as temporal bounds are not required for level E cores.

B. Implementing PWB partitioning and slack scheduling for non-critical write-back responses

To address the blocking interference due to a single per-core PWB buffer (Section V-B), CARP partitions the PWB into a *critical-PWB* and a *non-critical PWB* as shown in Figure 4c. Critical PWB enqueue write-back responses due to requests from critical cores, and non-critical PWB enqueue write-back responses due to requests from non-critical cores. For the example in Figure 3, the critical-PWB of c_0^A will have write-back responses for Y and Z, and the non-critical PWB of c_0^A will have write-back response for X. To ensure that critical cores are not blocked by write-back responses due to level E cores, the arbitration policy between requests and write-back responses is only applied to the PR buffer and *critical-PWB* partition as shown in Figure 4c.

At the start of a slack slot, we prioritize ready requests from levels C-D cores over write-back responses and requests from level E cores as levels C-D cores execute higher criticality tasks. If there are no ready requests from levels C-D cores, the critical and non-critical PWBs of all cores are checked, and a write-back request is scheduled if found. If no ready requests from levels C-D cores and write-back responses in the cores are found, the slack slot is allocated to a level E core. We use RR to arbitrate across requests from multiple level E cores.

VII. Latency Analysis

We derive the per-request worst-case latency (WCL) bound a core experiences when it accesses data. The WCL bound of the requesting core has three latency components: (1) latency to broadcast data request on the network (request latency), (2) latency for a remote core with an updated copy of the requested data and/or the shared memory to place the data response on the network (communication latency), and (3) the latency of the data response to arrive at the requesting core (response latency). The arbitration scheme determines the request and response latencies. The communication latency, however, depends on the simultaneous communication between other cores in the system on the requested data. The WCL bound is the summation of these latency components.

A. Preliminaries

We envision CARP to be deployed on prior MCS-specific arbitration schemes such as [14], [15] that either combine different arbitration policies (TDM and RR) [15] or allocate different number of slots based on the core's criticality level (weighted TDM) [14]. The key features of these arbitration schemes are: (1) differential service guarantees to cores based

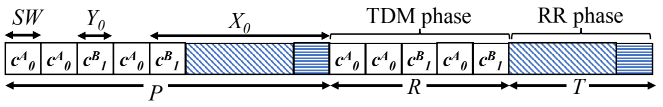


Fig. 5: Generalized MCS arbitration scheme [15].

TABLE III: Symbols used in latency analysis.

Symbol	Description
SW	Slot width
c_i^l	Core i running level l task
s_i	Slots allocated to core c_i^l
X_i	Slots between two allocated slots of c_i^l
Y_i	Slots between two allocated slots of c_i^l
T	Length of RR-phase (includes reserve slot)
R	Slots between two RR-phases
P	Arbitration period

on their criticality levels [14], [15], and (2) slack allocation for lower critical tasks [6], [33]. We capture these features in a representative arbitration scheme (shown in Figure 5), and use this scheme to derive the WCL bounds. Computing details such as the allocation of slots to cores is beyond the scope of this work; hence, these details are abstracted as arbitration parameters in the latency analysis.

The generic arbitration scheme consists of two arbitration phases: a weighted TDM arbitration phase (TDM-phase), and a RR arbitration phase (RR-phase). Levels A and B cores (C^{AB}) access the bus using TDM arbitration policy, and levels C and D cores (C^{CD}) access the bus using RR arbitration policy. We set the TDM slot width SW to be equal to completing one memory access. This slot width takes into account the latency to communicate coherence messages and data between cores and shared memory via the shared bus. We denote s_i as the number of slots allocated to c_i^l in the weighted TDM schedule. For example, $s_0 = 3$ and $s_1 = 2$ in Figure 5. For a core c_i^l, we use X_i to mean the maximum number of slots between its next dedicated slot, and Y_i as the *next* maximum number of slots between its next dedicated slot. A RR-phase consists of a sequence of time slots that are distributed to cores in a work-conserving RR manner. Once a core is granted access to the bus in a RR-phase, it relinquishes access when the request is completed or a threshold amount of time (SW) elapses when the core is granted access to the bus. We denote R as the number of slots between two RR-phases as shown in Figure 5. A RR-phase is augmented with a *reserve time slot* to avoid bus interference between cores accessing the bus in RR-phase and TDM-phase phases [15]. In-flight requests that accessed the bus in a RR-phase are allowed to complete in the reserve slot. However, new requests from cores are not allowed to access the bus in the reserve slot. The length of a RR-phase, which includes the reserve slot, is denoted as T. The key difference between slots in the RR-phase and TDM-phase is that cores can access the bus at any time instance in a RR-phase, whereas cores access the bus at the start of a slot in a TDM-phase. The arbitration period is denoted as P. Level E cores are granted access to the bus in slack. We denote the latency to transfer data from the shared memory to the core as L^{acc}. Table III summarizes the symbols used in the latency analysis.

Note that CARP is not only designed for MCS-specific arbitration schemes, and can also be deployed on a simple TDM arbitration scheme that allocates equal number of slots to all cores in the system. However, the latency analysis for levels C-D cores, and their impact on the analysis of levels A-B cores will change as the analysis in the following section assumes RR allocation for levels C-D cores.

B. Analysis

First, we derive the worst-case request and data response latencies of a request issued by a core under analysis c_{ua}^l based on its criticality level (Theorems 1-2). Then, we present a critical instance that results in the worst-case communication latency incurred by c_{ua}^l's request (Lemma 1). The critical instance identifies the memory access pattern of cores of different criticality levels that interfere with the data response of c_{ua}^l's request to X. From this critical instance, Lemma 2 derives the worst-case number of cores and their criticality levels that interfere with c_{ua}^l's request, and Theorems 3-4 derive the worst-case communication latencies of c_{ua}^l based on the criticality levels of the interfering cores. The WCL bounds are computed for criticality levels A-D. CARP does not provide any WCL bounds for level E cores; hence, we do not derive their WCL bounds.

Theorem 1. *The worst-case request latency for c_{ua}^l to X is given by:*

$$WCL^{Req}(c_{ua}^l) = \begin{cases} (2 + X_{ua} + Y_{ua}) \times SW & : \textit{if } l \in \{A, B\} \\ 2 \times \Big(\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1 + R) & \\ \qquad + |C^{\mathsf{CD}}| - 1 \Big) \times SW & : \textit{if } l \in \{C, D\} \end{cases}$$

Proof. Consider the following two cases. The first case is when $l \in \{A, B\}$, and the second case is when $l \in \{C, D\}$. Suppose c_{ua}^l such that $l \in \{A, B\}$ accesses the shared bus during its pre-allocated TDM slot. In the worst-case, c_{ua}^l attempts to broadcast a request immediately after the start of its TDM slot. Since requests must be broadcasted on the bus at the start of the slot, c_{ua}^l can only successfully broadcast its request during its next pre-allocated TDM slot. Thus, c_{ua}^l must wait for the duration of its own slot whose start it just missed, and all other dedicated slots preceding its own next slot in the arbitration schedule (X_{ua}) resulting in a latency of $(1 + X_{ua})$ slots. In the worst-case, this slot is allocated for servicing write-back responses. As a consequence, c_{ua}^l must wait an additional $(1 + Y_{ua})$ for its next slot to broadcast its request. Therefore, in the worst-case, the broadcast request latency for c_{ua}^l is $(2+X_{ua}+Y_{ua})$ slots. Suppose c_{ua}^l such that $l \in \{C, D\}$ accesses the shared bus. The worst-case scenario occurs when c_{ua}^l attempts to broadcast a request when all other $(|C^{\mathsf{CD}}|-1)$ cores have pending requests. Hence, c_{ua}^l is blocked from successfully broadcasting its request while the $|C^{\mathsf{CD}}|-1$ pending requests are serviced. These $|C^{\mathsf{CD}}| - 1$ cores can access the bus over multiple round-robin arbitration rounds, which we compute by $\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil$. Therefore, c_{ua}^l must wait for $(\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil)$ number of reserve slots, and $(\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil)$ number of R slots, which comprises of TDM slots of C^{AB} cores. After this delay, c_{ua}^l receives a slot to issue its request. However, in the worst-case, this slot may be used to service c^l_{ua}'s write-back response. Consequently, c^l_{ua} must wait an additional latency of $(\lceil\frac{|C^{\mathsf{CD}}|}{T}\rceil\times(1+R)+|C^{\mathsf{CD}}|-1)$ slots resulting in a total broadcast request latency of $2\times\left(\lceil\frac{|C^{\mathsf{CD}}|}{T}\rceil\times(1+R)+|C^{\mathsf{CD}}|-1\right)$. □

Theorem 2. *The worst-case response latency for c^l_{ua} to receive* X *is given by:*

$$WCL^{Resp}(c^l_{ua}) = WCL^{Req}(c^l_{ua}) + L^{acc}$$

Proof. Data responses are also sent from shared memory at the start of the receiving core's slot. As a result, the worst-case response latency is equal to the worst-case request latency and L^{acc}. We omit the proof of this theorem as it is similar to the proof of Theorem 1. □

Lemma 1. *The worst-case communication latency of c^l_{ua} where $l \in \{A, B, C, D\}$ when data is communicated across criticality levels occurs when c^l_{ua} issues a read or write request to line* X *such that (1) α levels A and B cores broadcast write request to* X *before c^l_{ua}'s request is broadcasted, and (2) β levels C and D cores broadcast write requests to* X *before c^l_{ua}'s request is broadcasted.*

Proof. There are two cases to consider. The first case proves by contradiction that a scenario in which at least one A-D core broadcasts a read request instead of a write request to X does not result in the worst-case communication latency. The second case proves by contradiction that a scenario in which at least one A-D core broadcasts a write request after c^l_{ua} broadcasts its request to X does not result in the worst-case interference latency. We use Figure 6 to assist in the readability of the proof by contrasting these cases with the worst-case scenario for a 4-core MCS multi-core platform.

Suppose $\exists\ c^l_i$ where $l \in \{\mathsf{A,B,C,D}\}$ that broadcasts a read request to X instead of a write request. Recall from Section III that a read request from one core does not require write-backs from other cores (SWMR invariant). As a result, c^l_{ua} will not experience any interference from c^l_i. Hence, c^l_{ua}'s data response will incur communication latency from $\alpha+\beta-1$ cores, which is less than that from $\alpha+\beta$ cores. As an example, consider the following modification to the worst-case scenario depicted in Figure 6: c^C_2 broadcasts a read request to X instead of a write request. c^C_2 receives X in slot 16. Since c^C_2's request is a read, and does not modify X, c^D_3 can receive X and complete its write request in slot 17. c^A_0 waits for c^D_3 to complete the write-back for X, and receives X in slot 24, which is less than the worst-case instance (30 slots).

Suppose $\exists\ c^l_i$ where $l \in \{\mathsf{A,B,C,D}\}$ that broadcasts a write request after c^l_{ua} broadcasts its request to X. As a result, c^l_i will receive X *after* c^l_{ua} receives X. Hence, c^l_{ua}'s data response will not incur any interference from c^l_i resulting in communication latency less than that when $|\alpha(l)| + |\beta(l)|$ cores broadcast write requests before c^l_{ua} broadcasts its request to X. As an example, consider the following modification to worst-case scenario depicted in Figure 6: c^B_1 broadcasts its write request to X after c^A_0 broadcasts its read request to X. Therefore, c^A_0 receives X before c^B_1 in slot 24, and does not incur the latency of c^B_1 to do a write-back resulting in a lower WCL. □

Lemma 2. *For c^l_{ua} where $l \in \{A, B, C, D\}$, the maximum number of level A-B cores ($\alpha(l)$), and the maximum number of level C-D cores ($\beta(l)$) that can broadcast requests before c^l_{ua} broadcasts its request is given by:*

$$\alpha(l) = \begin{cases} |C^{\mathit{AB}} \setminus \{c^l_{ua}\}| & : \textit{if } l \in \{A,B\} \\ |C^{\mathit{AB}}| & : \textit{if } l \in \{C,D\} \end{cases}$$

$$\beta(l) = \begin{cases} min(|C^{\mathit{CD}}|, \lceil\frac{WCL^{Req}(c^l_{ua})}{P}\rceil\times(T-1) \\ \qquad +max(0,(O-|C^E|))) & : \textit{if } l \in \{A,B\} \\ |C^{\mathit{CD}} \setminus \{c^l_{ua}\}| & : \textit{if } l \in \{C,D\} \end{cases}$$

where O is the total number of slack slots in $WCL^{Req}(c^l_{ua})$, and is computed as
$O = WCL^{Req}(c^l_{ua}) - \lceil\frac{WCL^{Req}(c^l_{ua})}{P}\rceil\times T - \alpha.$

Proof. Suppose $l \in \{\mathsf{A,B}\}$. From Theorem 1, WCL^{Req}_{ua} comprises of X_{ua} slots in which $|C^{\mathsf{AB}}|-1$ cores can broadcast their requests to X. Hence, $|\alpha(l)| = |C^{\mathsf{AB}}\setminus\{c^l_{ua}\}|$ when $l \in \{\mathsf{A,B}\}$. Suppose $l \in \{\mathsf{C,D}\}$. From Theorem 1, WCL^{Req}_{ua} comprises of at least R slots in which $|C^{\mathsf{AB}}|$ cores can broadcast their requests to X Hence, $|\alpha(l)| = |C^{\mathsf{AB}}|$ when $l \in \{\mathsf{C,D}\}$.
Suppose $l \in \{\mathsf{A,B}\}$. WCL^{Req}_{ua} comprises of $\lceil\frac{WCL^{\mathrm{Req}}_{ua}}{P}\rceil$ round-robin arbitration phases for C^{CD} cores to broadcast requests. Since the round-robin arbitration phase is of length T slots, and consists of one reserve slot where cores are allowed to complete pending requests but cannot broadcast requests, $\lceil\frac{WCL^{\mathrm{Req}}_{ua}}{P}\rceil\times(T-1)$ level C-D cores can broadcast requests in WCL^{Req}_{ua}. Furthermore, we allow C^{CD} cores to broadcast requests in slack slots that are not utilized by C^E cores. The total number of slack slots is denoted as O. If $O-|C^E| > 0$, then there are slack slots that can be utilized by C^{CD} cores to broadcast requests. If $O-|C^E| \le 0$, then there are enough C^E cores to utilize the slack slots. The maximum number of C^{CD} cores that can broadcast requests to X is bounded by $|C^{\mathsf{CD}}|$.
Suppose $l \in \{\mathsf{C,D}\}$. WCL^{Req}_{ua} comprises of $|C^{\mathsf{CD}}|$ slots where all remaining $|C^{\mathsf{CD}}|-1$ receive a slot to broadcast requests. As a result, $|C^{\mathsf{CD}}|\setminus\{c^l_{ua}\}$ cores can broadcast requests before c^l_{ua} broadcasts its request. □

Theorem 3. *The worst-case communication latency incurred by c^l_{ua}'s request to* X *where $l \in \{A, B\}$ due to c^m_j's interfering write request to* X *is given by:*

$$WCL^{Comm}_{l,m} = \begin{cases} \lceil\frac{2}{s_j}\rceil\times P\times SW & : \textit{if } m \in \{A,B\} \\ (2\times\lceil\frac{|C^{CD}|}{T}\rceil)\times P\times SW & : \textit{if } m \in \{C,D\} \end{cases}$$

Proof. c^m_j requires two allocated slots after broadcasting its request to receive the data for X, and write-back X. This is because after c^m_j broadcasts its request to X, c^m_j receives X from memory, and must write-back X in the next slot resulting in a latency of 2 allocated slots of c^m_j. Suppose c^m_j such that $m \in \{\mathsf{A,B}\}$. Since c^m_j requires two slots to receive, update, and write-back X, the total communication latency due to c^m_j's request to X is $\lceil\frac{2}{s_j}\rceil\times P$ slots.

Suppose c^m_j such that $m \in \{\mathsf{C,D}\}$. Since c^m_j is granted access to the bus using RR arbitration policy, c^m_j must wait

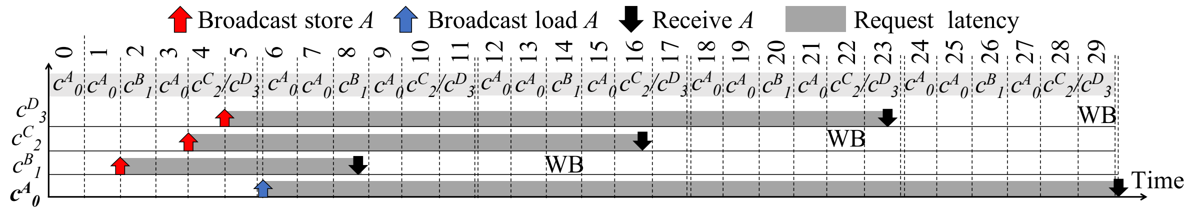


Fig. 6: Worst-case instance for a 4-core system. c^l_{ua} is c^A_0.

$(\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil - 1) \times P$ slots to receive X. Hence, the total communication latency due to c^m_j's request to X is $(2 \times \lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil) \times P$ slots. □

Theorem 4. *The worst-case communication latency incurred by c^l_{ua}'s request to* X *where $l \in \{C, D\}$ due to c^m_j's interfering write request to* X *is given by:*

$$WCL^{Comm}_{l,m} = \begin{cases} 2 \times \Big(\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1+R) + \\ \qquad |C^{\mathsf{CD}}| \Big) \times SW & : \textit{if } m \in \{A, B\} \\ 3 \times \Big(\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1+R) + \\ \qquad |C^{\mathsf{CD}}| \Big) \times SW & : \textit{if } m \in \{C, D\} \end{cases}$$

Proof. Suppose c^m_j such that $m \in \{\mathsf{A}, \mathsf{B}\}$. Recall that a core requires two allocated slots to receive and modify X, and write-back X. As a result, c^m_j requires $\lceil \frac{2}{s_j} \rceil \times P$ slots to complete these operations on X. Since R denotes the number of slots between two successive round-robin arbitration phases in the schedule, and $|C^{\mathsf{CD}}| > T$, $2 \times (\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1+R) + |C^{\mathsf{CD}}|) > \lceil \frac{2}{s_j} \rceil$. Hence, c^l_{ua} must wait $2 \times (\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1+R) + |C^{\mathsf{CD}}|)$ slots to receive X. Suppose c^m_j such that $m \in \{\mathsf{C}, \mathsf{D}\}$. When $|C^{\mathsf{CD}}| > T$, in the worst-case, c^m_j is allocated one slot in every round-robin arbitration phase. As a result, c^m_j requires $2 \times (\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1+R) + |C^{\mathsf{CD}}|)$ slots to receive and modify X, and write-back X. Since c^l_{ua} is also allocated one slot in every round-robin arbitration phase in the worst-case, c^l_{ua} must wait $3 \times (\lceil \frac{|C^{\mathsf{CD}}|}{T} \rceil \times (1+R) + |C^{\mathsf{CD}}|)$ slots to receive X. □

Theorem 5. *The total worst-case latency incurred by c^l_{ua}'s request to* X *where $l \in \{A, B, C, D\}$ is given by:*

$$WCL^{Total}(c^l_{ua}) = \begin{array}{l} WCL^{Req}(c^l_{ua}) + \sum_{i \in \alpha(l) + \beta(l)} WCL^{Comm}_{l,i} \\ + WCL^{Resp}(c^l_{ua}) \end{array}$$

Proof. The total WCL of c^l_{ua}'s request includes the worst-case request and data response latencies, and the worst-case communication latency due to interfering cores $|\alpha(l)| + |\beta(l)|$. □

C. Discussion

A key distinguishing feature between the WCL bounds of PMSI [41] and CARP is that WCL bounds in CARP are *independent* of the number of level E cores. On the other hand, WCL bounds in PMSI increase with level E cores as PMSI *elevates* level E cores to critical cores. Hence, CARP provides *tighter* WCL bounds for critical cores compared to PMSI.

The above analysis derives the *per-request* WCL bound to shared data that has *both* read and write permissions. On the other hand, the WCL bound of a request to shared data that has *only* read permissions is SW cycles as there are no coherence transitions for read-only shared data. Using static analysis tools, we envision the following process that utilizes the derived WCL bounds to compute the end-to-end WCET of a task. Static analysis tools can provide information on (1) the read/write patterns to a memory address and (2) the number of cores accessing a memory address. Based on the read/write patterns to a memory address, the appropriate WCL bound can be used, and the number of cores accessing a memory address can be used to substitute the parameters α and β derived previously in the analysis. Hence, the WCET of the task can be derived by applying this procedure to the memory addresses accessed by the task.

TABLE IV: Hybrid arbitration policy parameters.

Parameter	Value
P	15 slots
X_0, X_1, X_2, X_3	11, 11, 13, 13 slots
Y_0, Y_1, Y_2, Y_3	0 slots
R	12 slots
T	3 slots
SW	50 cycles
$\lvert\alpha(\mathsf{A/B})\rvert$, $\lvert\alpha(\mathsf{C/D})\rvert$	3 cores, 4 cores
$\lvert\beta(\mathsf{A/B})\rvert$, $\lvert\beta(\mathsf{C/D})\rvert$	2 cores, 2 cores

VIII. Methodology

We use gem5 [47] to evaluate CARP. gem5 is a micro-architectural simulator that models the memory subsystem and coherence protocol with high precision. CARP is simulated on a multi-core platform that comprises of 8 cores (c_0-c_7) running at 2GHz. Our simulated multi-core platform does not run an OS. The cores implement in-order pipelines, and cores can have a single pending memory request. We allocate the following criticalities to the cores: c^A_0, c^A_1, c^B_2, c^B_3, c^C_4, c^C_5, c^D_6, c^E_7. Note that these allocations to cores are only done for empirical evaluation, a different mapping is also possible. Each core has a private L1 32KB 4-way instruction cache and 32KB 4-way data cache. The access latency to each private cache is set to 3 cycles. All cores share a 1MB set associative last-level cache (LLC). We configure the LLC such that all LLC accesses are hits in order to isolate and focus on the impact of maintaining cache coherence on the shared data access latencies. We set the LLC access latency to 50 cycles. Both the private L1 caches and shared LLC operate on cache line sizes of 64 bytes. The cores and the shared LLC are connected via a shared snooping bus that deploys an instance of the generalized arbitration policy described in Table IV.

We evaluate CARP against prior data communication mechanisms proposed for multi-core real-time and MCS platforms

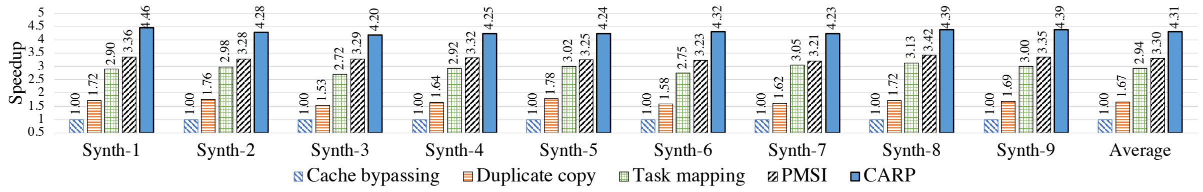


Fig. 7: Performance of design choices and CARP on synthetic workloads.

TABLE V: Observed WCL for synthetic benchmarks.

Level	Analytical WCL (cycles)	Observed WCL (cycles)
A	6600	4348
B	6800	3701
C/D	11200	6699

such as (1) *duplication* of communicated data [7], [48], (2) *cache bypassing* of communicated data [7], [49], [50], (3) *mapping* communicating tasks to the same core [6], and (4) the recently proposed *PMSI cache coherence protocol* [41]. For the PMSI cache coherence protocol, we elevate level E cores to critical cores as PMSI was designed for multi-core real-time systems where all cores are of the same criticality level [41]. We also evaluate CARP against MSI and MESI conventional cache coherence protocols [42]. Prior work showed that that deploying conventional coherence protocols on a predictable bus arbitration scheme can result in unbounded latencies for shared data accesses [41]. Hence, the conventional MSI and MESI protocols are executed on a snooping bus that does not deploy a predictable arbitration policy. We do not evaluate CARP against HourGlass [28] as it does not deal with tasks of varying criticality levels, and does not provide enough guidance on setting timer values.

Our evaluation uses synthetic benchmarks and SPLASH-2 [29], a multi-threaded benchmark suite. In the synthetic benchmarks (Synth1-Synth9), all cores except c_7^{E} perform the *same* sequence of operations (read/write) on shared data. c_7^{E} only performs read operations on shared data. As a result, these benchmarks stress the states and transitions of CARP. The synthetic benchmarks vary with each other based on the proportion of read and write operations. For these benchmarks, we run our simulation for 100,000 total memory operations across all cores. The SPLASH-2 benchmark suite consists of multi-threaded benchmarks derived from various domains such as graphics rendering, and scientific computation [29]. We use the SPLASH-2 benchmark suite due to a lack of available multi-threaded applications that operate on shared data, and are representative of those used in MCS. We run all SPLASH-2 benchmarks until completion. We used the SPLASH-2 benchmark to verify the data correctness of CARP, and observed that all benchmarks terminated with correct data output when executed using CARP.

IX. Results

A. Synthetic workloads

Observed WCL. Table V shows the per-request observed total WCL across synthetic benchmarks deployed on CARP. CARP guarantees that the observed WCL are within the computed WCL bounds for critical cores across all benchmarks. In all benchmarks, the maximum observed request, data response, and communication latency components are within their respective analytical WCL bounds. We observe that the maximum request, and data response are equal to the respective analytical WCL bounds, and the maximum observed communication latencies vary across benchmarks as they are dependent on the memory activity on shared data. Benchmarks do not reach the maximum communication latency derived from the analysis because cores complete their requests earlier due to slack slots, and work conserving RR arbitration policies.

Average-case performance. Figure 7 shows the speedup in total execution time for the synthetic benchmarks using the design choices described earlier and CARP. We normalize the execution time to the cache bypassing data communication mechanism. Duplicating communicated data and mapping tasks to the same core offer better performance over private cache bypassing as they allow communicated data to reside in the private caches. As a result, these techniques offer 1.67x and 2.94x performance speedup over cache bypassing. Note that duplicating communicated data increases the arbitration period as it requires level A-D cores to communicate updates to communicated data to the duplicate copies, and hence, does not perform as well as the task mapping technique. Mechanisms that use cache coherence (PMSI and CARP) offer better performance over task co-location technique as these mechanisms do not restrict task parallelism, and allow multiple copies of communicated data to reside in the private caches of all cores. CARP performs better than PMSI (30% on average) due to slack allocation for level E cores and the MCS arbitration schedule. Unlike PMSI, CARP is deployed on an arbitration policy that allocates different number of slots to cores based on their criticality levels. As a result, level A-D communicate more than one data in an arbitration period, which improves their communication throughput.

Performance slowdown relative to MSI and MESI. Conventional coherence protocols (MSI and MESI) are designed for average-case performance, and are not designed to be predictable [41], [51]. These protocols are not deployed on a predictable bus arbitration mechanism. As a result, a core can broadcast its request as soon as the CC generates the request or immediately complete a write-back response on observing a remote write request. CARP exhibits an average performance slowdown of 73% and 66% compared to the MESI and MSI cache coherence protocols respectively. We find this slowdown reasonable for achieving predictability.

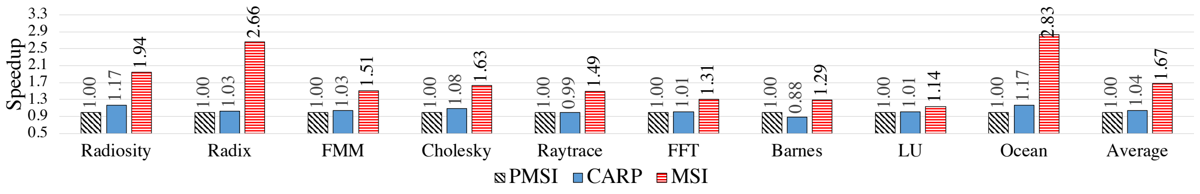


Fig. 8: Performance of CARP on SPLASH-2.

B. *SPLASH-2 workloads*

For SPLASH-2 workloads, we run CARP with level A-D cores and no level E cores. This is because, in SPLASH-2 workloads, *all* launched threads read and write on shared data structures such as locks and conditional barriers during execution in order to maintain benchmark correctness. We confirmed that the observed WCL bounds for level A-D cores are within the analytical bounds (not shown). Figure 8 shows the performance speedup of CARP compared to PMSI and the conventional MSI cache coherence protocol. The results are normalized to PMSI. We observe that CARP offers a performance improvement of 4% over PMSI, and shows an average slowdown of 60% compared to the conventional MSI protocol. In these benchmarks, thread barriers force all threads to converge at the barrier before making forward progress. As a consequence, level A and B cores that may benefit from the additional allocated slots end up waiting for level C and D cores that have fewer allocated slots resulting in increased execution time.

X. Related works

There has been limited attention towards designing predictable data communication between tasks of different criticality levels in a multi-core MCS platform [1], [6]. Chisholm et al. [6] proposed a technique to allow data communication between levels A-C tasks in a multi-core MCS platform. Their technique applied mechanisms such as cache bypassing, mapping communicating tasks to one core, cache locking, and data duplication to enable task communication. As a result, their technique relies on OS and hardware support. In contrast, CARP is a hardware technique that does not restrict the usage of caches, does not require a particular mapping of tasks to cores, and does not duplicate communicated data. Becker et al. [1] proposed an alternative approach that constructed offline schedules of computation and memory phases of tasks such that contending data communication between multiple tasks were not scheduled at the same time. This approach relied on the availability of memory and compute details of the real-time tasks. CARP on the other hand, does not require information about the compute and memory behavior of the tasks.

Prior data communication techniques for multi-core real time systems used three main approaches: 1) disabled caching of communicated data in private caches [6], [49], [50], 2) replicated communication data [48], and 3) scheduled tasks that communicate data with each other on the same core through OS changes [6], [48], [52], [53]. These approaches provided predictable data sharing at the cost of reduced average-case performance. Alternatively, Pyka et al. [54] modified the application such that data communication on the same data were protected using software locks. The effect was that only one core performed data communication at any time instance. CARP on the other hand allows multiple cores to carry out their data communication simultaneously. Hassan et al. [41] recently described a set of guidelines for predictable data communication between real-time tasks using hardware cache coherence, and proposed PMSI, a predictable cache coherence protocol to that was built on these guidelines. CARP is also built using these design guidelines, and includes additional features that enable CARP to be criticality-aware, and satisfy requirements specific to MCS. Sritharan et al. [28] recently proposed HourGlass, a time-based cache coherence protocol for dual-critical MCS systems. In this protocol, cores retain lines in their private caches for a duration of time period irrespective of the remote memory activity on the same lines, and the lines self-invalidate after the time period. The key novelty in HourGlass was that the time periods for a line were configured based on (1) the owner's criticality level and (2) the remote cores' criticality levels that issued memory requests to the same line. As a consequence, in HourGlass, memory activity of non-critical cores on shared data contributed to the WCL bounds of critical cores [28], which further requires an extension to the safety guidelines put forward in the ISO-26262 and AUTOSAR standards. On the other hand, CARP is designed to allow for data communication between critical and non-critical cores such that memory activity of non-critical cores do not contribute to the WCL bounds of critical cores.

Recently, Sensfelder et al. [55] provided a formal framework to model and analyze the latency interference in conventional bus based cache coherence protocols. We are motivated to apply similar models to CARP in order to formally verify the sources of interference due to data sharing in MCS, and reserve this exploration for future work.

XI. Conclusion

As MCS platforms continue to integrate multiple complex tasks of varying criticality levels, enabling data communication between these tasks will be essential to realize added functionality and improve the overall responsiveness of the MCS. In this work, we present CARP, a criticality-aware hardware cache coherence protocol to realize predictable and high performance data communication between tasks executing on a multi-core MCS without violating the safety requirements of critical tasks. Our evaluation of CARP using synthetic and real-world benchmarks show that CARP guarantees the safety requirements of critical tasks, and improves average-case performance of MCS compared to prior techniques.

XII. Acknowledgement

We would like to thank the anonymous reviewers for their valuable feedback and suggestions.

References

[1] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nlis, and T. Nolte, "Contention-free execution of automotive applications on a clustered many-core platform," in *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 14–24, 2016.

[2] M. Jung, S. A. McKee, C. Sudarshan, C. Dropmann, C. Weis, and N. Wehn, "Driving into the memory wall: The role of memory for advanced driver assistance systems and autonomous driving," in *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, (New York, NY, USA), pp. 377–386, ACM, 2018.

[3] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *European Dependable Computing Conference*, pp. 132–143, IEEE, 2012.

[4] A. Burns and R. I. Davis, "A Survey of Research into Mixed Criticality Systems," *ACM Computing Surveys*, pp. 82:1–82:37, 2018.

[5] S. Vestal, "Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance," in *Real-Time Systems Symposium (RTSS)*, pp. 239–243, IEEE, Dec 2007.

[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 *Real-Time Systems Symposium (RTSS)*, pp. 57–68, IEEE, Nov 2016.

[7] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst, "Communication Centric Design in Complex Automotive Embedded Systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 10:1–10:20, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[8] E. Bost, "Hardware support for robust partitioning in freescale qoriq multicore socs (p4080 and derivatives)," *Freescale Semiconductor, Inc., Tech. Rep.*, 2013.

[9] C. E. Salloum, M. Elshuber, O. Hftberger, H. Isakovic, and A. Wasicek, "The ACROSS MPSoC – A New Generation of Multi-core Processors Designed for Safety-Critical Embedded Systems," in *Euromicro Conference on Digital System Design*, pp. 105–113, IEEE, Sept 2012.

[10] B. D. de Dinechin, R. Ayrignac, P. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel, "A clustered manycore processor architecture for embedded and accelerated applications," in *High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, Sep. 2013.

[11] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, *et al.*, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, pp. 449–471, 2015.

[12] NVIDIA, "JETSON TK1: Unlock the power of the GPU for embedded systems applications," 2016.

[13] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality," in *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 299–308, IEEE Computer Society, 2012.

[14] M. Hassan and H. Patel, "Criticality- and Requirement-Aware Bus Arbitration for Multi-Core Mixed Criticality Systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 1–11, IEEE, April 2016.

[15] B. Cilku, A. Crespo, P. Puschner, J. Coronel, and S. Peiro, "A TDMA-Based arbitration scheme for mixed-criticality multicore platforms," in *International Conference on Event-based Control, Communication, and Signal Processing (EBCCSP)*, IEEE, June 2015.

[16] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Scheduling of Mixed-criticality Applications on Resource-sharing Multicore Systems," in *International Conference on Embedded Software (EMSOFT)*, pp. 17:1–17:15, ACM, 2013.

[17] A. Kritikakou, C. Pagetti, O. Baldellon, M. Roy, and C. Rochange, "Run-time control to increase task parallelism in mixed-critical systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 119–128, IEEE, July 2014.

[18] D. Guo and R. Pellizzoni, "A requests bundling DRAM controller for mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 247–258, IEEE, 2017.

[19] M. Hassan, H. Patel, and R. Pellizzoni, "PMC: A Requirement-Aware DRAM Controller for Multicore Mixed Criticality Systems," *ACM Transactions on Embedded Computing Systems*, pp. 100:1–100:28, 2017.

[20] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh, "A predictable and command-level priority-based DRAM controller for mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 317–326, IEEE, April 2015.

[21] P. K. Gadepalli, G. Peach, G. Parmer, J. Espy, and Z. Day, "Chaos: a System for Criticality-Aware, Multi-core Coordination," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, 2019.

[22] AUTOSAR, "Overview of functional safety measures in autosar," vol. 4.4.0, 2018.

[23] AUTOSAR, "Autosar model constraints," vol. 4.3.1, 2017.

[24] X. Zheng, C. Julien, R. Podorozhny, and F. Cassez, "BraceAssertion: Runtime Verification of Cyber-Physical Systems," in *International Conference on Mobile Ad Hoc and Sensor Systems*, pp. 298–306, IEEE, Oct 2015.

[25] X. Zheng, C. Julien, R. Podorozhny, F. Cassez, and T. Rakotoarivelo, "Efficient and Scalable Runtime Monitoring for CyberPhysical System," *IEEE Systems Journal*, pp. 1667–1678, June 2018.

[26] C. W. W. Wu, D. Kumar, B. Bonakdarpour, and S. Fischmeister, "Reducing monitoring overhead by integrating event- and time-triggered techniques," in *Runtime Verification*, pp. 304–321, Springer Berlin Heidelberg, 2013.

[27] I. O. for Standardization, "ISO 26262," 2011.

[28] N. Sritharan, A. M. Kaushik, M. Hassan, and H. D. Patel, "Hourglass: Predictable time-based cache coherence protocol for dual-critical multi-core systems," *CoRR, abs/1706.07568*, 2017.

[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *International Symposium on Computer Architecture (ISCA)*, ISCA '95, (New York, NY, USA), pp. 24–36, ACM, 1995.

[30] Federal Aviation Administration (FAA), "Position Paper CAST-32A," 2016.

[31] RTCA Inc., "Software Considerations in Airborne Systems and Equipment Certification," 1992.

[32] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," in *International Conference on Computer and Information Technology (CIT)*, (Washington, DC, USA), pp. 1864–1871, IEEE Computer Society, 2010.

[33] F. Hebbache, M. Jan, F. Brandner, and L. Pautet, "Shedding the shackles of time-division multiplexing," in *Real-Time Systems Symposium (RTSS)*, pp. 456–468, IEEE, Dec 2018.

[34] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens, "A Globally Arbitrated Memory Tree for Mixed-Time-Criticality Systems," *IEEE Transactions on Computers*, pp. 212–225, Feb 2017.

[35] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury, "Static analysis of multi-core TDMA resource arbitration delays," *Real-Time Systems*, pp. 185–229, 2014.

[36] M. Yoon, J. Kim, and L. Sha, "Optimizing Tunable WCET with Shared Resource Allocation and Arbitration in Hard Real-Time Multicore Systems," in *Real-Time Systems Symposium (RTSS)*, pp. 227–238, IEEE, Nov 2011.

[37] M. Schoeberl, W. Puffitsch, S. Hepp, B. Huber, and D. Prokesch, "Patmos: A Time-predictable Microprocessor," *Real-Time Systems*, pp. 389–423, 2018.

[38] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 99–108, IEEE/ACM, Oct 2011.

[39] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo, "Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms," in *Euromicro Conference on Real-Time Systems (ECRTS)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[40] M. Hassan, "Heterogeneous MPSoCs for Mixed-Criticality Systems: Challenges and Opportunities," *IEEE Design Test*, pp. 47–55, 2018.

[41] M. Hassan, A. M. Kaushik, and H. Patel, "Predictable Cache Coherence for Multi-core Real-Time Systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 235–246, IEEE, 2017.

[42] D. J. Sorin, M. D. Hill, and D. A. Wood, “A primer on memory consistency and cache coherence,” *Synthesis Lectures on Computer Architecture*, 2011.

[43] A. Cortex, “Cortex-A9 MPCore,” *Technical Reference Manual*, 2009.

[44] N. Kurd, P. Mosalikanti, M. Neidengard, J. Douglas, and R. Kumar, “Next generation Intel® core micro-architecture (nehalem) clocking,” *IEEE Journal of Solid-State Circuits*, 2009.

[45] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange, “Autosar–a worldwide standard is on the road,” in *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, vol. 62, p. 5, 2009.

[46] A. Kostrzewa, S. Saidi, and R. Ernst, “Slack-based Resource Arbitration for Real-time Networks-on-chip,” in *Design, Automation & Test in Europe (DATE)*, pp. 1012–1017, EDA Consortium, 2016.

[47] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The Gem5 Simulator,” *SIGARCH Computer Architecture News*, pp. 1–7, 2011.

[48] N. Kim, M. Chisholm, N. Otterness, J. H. Anderson, and F. D. Smith, “Allowing shared libraries while supporting hardware isolation in multicore real-time systems,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 223–234, IEEE, 2017.

[49] D. Hardy, T. Piquet, and I. Puaut, “Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches,” in *Real-Time Systems Symposium (RTSS)*, pp. 68–77, IEEE, 2009.

[50] B. Lesage, D. Hardy, and I. Puaut, “Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures.,” in *18th International Conference on Real-Time and Network Systems*, (Toulouse, France), p. 2283, Nov. 2010.

[51] G. Gracioli and A. A. Frhlich, “On the influence of shared memory contention in real-time multicore applications,” in *Brazilian Symposium on Computing Systems Engineering*, pp. 25–30, Nov 2014.

[52] J. M. Calandrino and J. H. Anderson, “On the Design and Implementation of a Cache-Aware Multicore Real-Time Scheduler,” in *Euromicro Conference on Real-Time Systems (ECRTS))*, pp. 194–204, IEEE, 2009.

[53] G. Gracioli and A. A. Fröhlich, “On the Design and Evaluation of a Real-Time Operating System for Cache-Coherent Multicore Architectures,” *ACM SIGOPS Operating Systems Review - Special Topics*, vol. 49, pp. 2–16, Jan. 2016.

[54] A. Pyka, M. Rohde, and S. Uhrig, “Extended performance analysis of the time predictable on-demand coherent data cache for multi- and many-core systems,” in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pp. 107–114, IEEE, July 2014.

[55] N. Sensfelder, J. Brunel, and C. Pagetti, “Modeling Cache Coherence to Expose Interference,” in *Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 18:1–18:22, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.