PREPRINT: "M. Hassan and H. Patel, "Criticality- and Requirement-aware Bus Arbitration for Multi-core Mixed Criticality Systems," in proceed ings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016, pp. 1-11."

Criticality- and Requirement-aware Bus Arbitration for Multi-core Mixed Criticality Systems

Mohamed Hassan
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada.
Email: mohamed.hassan@uwaterloo.ca

Hiren Patel
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada.
Email: hiren.patel@uwaterloo.ca

***Abstract*—This work presents CArb, an arbiter for controlling accesses to the shared memory bus in multi-core mixed criticality systems. CArb is a requirement-aware arbiter that optimally allocates service to tasks based on their requirements. It is also criticality-aware since it incorporates criticality as a first-class principle in arbitration decisions. CArb supports any number of criticality levels and does not impose any restrictions on mapping tasks to processors. Hence, it operates in tandem with existing processor scheduling policies. In addition, CArb is able to dynamically adapt memory bus arbitration at run time to respond to increases in the monitored execution times of tasks. Utilizing this adaptation, CArb is able to offset these increases; hence, postpones the system need to switch to a degraded mode. We prototype CArb, and evaluate it with an avionics case-study from Honeywell as well as synthetic experiments.**

I. Introduction

Mixed-criticality systems (MCS) consist of a set of interacting software components, where the components may operate under various criticality levels (CLs) [1]. Each CL provides a degree of assurance against the software component's failure [2]. For instance, DO178C used in avionics denotes five CLs ranging from critical to no effect. The real-time research community is interested in using multi-cores to deploy MCS, mainly because multi-cores offer small sized, low weighted, and low-cost hardware platforms that are mainstream nowadays. However, this requires consolidating software components onto the multi-core platform, which implies sharing hardware resources such as processors, buses, caches and main memories among these components. Resource sharing brings out a key challenge in the design of MCS: to effectively schedule shared hardware resources so as to ensure safety guarantees mandated by the CLs, and to deliver the performance demanded by each software component.

Recent efforts addressing this challenge have focused on proposing models and scheduling algorithms that schedule tasks with CLs onto cores [3]–[7]. Earlier approaches proposed methods to deploy MCS onto single core platforms [3], [4], which were further advanced to multi-core platforms [5]–[7]. These efforts developed a standard model for MCS, where each task is characterized by a criticality level, usually two CLs: LO and HI. Each task has a worst-case execution time (WCET)estimate, S, for each CL, $S(LO)$ and $S(HI)$ for the two levels case. The system operates initially in a normal mode, where it considers the $S(LO)$ of each task and both higher- and lower-critical tasks utilize the hardware resources. If a critical task exceeds its $S(LO)$, the system switches to a degraded mode, where it suspends all lower-critical tasks and considers the $S(HI)$ of the higher-critical ones [8]. These dynamic migrations between various modes is a key characteristic of MCS as compared to single-criticality traditional real-time systems. Since this model evolved initially for single-core MCS, it suffers from two crucial weaknesses when applied to multi-core MCS. 1) As observed by [9], approaches adopting this model do not incorporate inter-task interferences arising from accessing resources that are shared amongst cores such as memory buses, caches, and main memories in their scheduling or analysis. Experiments show that memory interferences can contribute up to 300% to the WCET of a task [10], while the memory bus interference in commercial-off-the-shelf (COTS) systems can solely increase the WCET up to 44% [11]. As a consequence, we find it is of unavoidable necessity to account for these interferences for multi-core MCS. 2) These approaches, upon switching to the degraded mode, do not provide any guaranteed service to lower-critical tasks. Since lower-critical tasks are still critical, industry criticizes this action as it may result in safety issues [2].

Fortunately, recent works address interference in MCS due to shared dynamic random access memories (DRAMs) [12], [13] and shared caches amongst cores [14], [15]. Nonetheless, there is a limited focus on addressing the interference problem on shared buses in MCS. To our knowledge, [16] [9] are the only approaches to incorporate the memory bus interference in MCS modelling. However, both have certain limitations. [16] is adequate only for two criticality levels and mandates a particular mapping of tasks to cores. [9] comprises predictable COTS bus arbiters such as round-robin (RR) and first-come-first-serve (FCFS), which lack the criticality notion. As a result, we find that the obtained bounds in [9] are pessimistic, foremost because of lacking a criticality-aware arbitration amongst different traffics on the memory bus. In addition, we find that these limitations in [9], [16] disallow them from exploring possible novel solutions at the arbiter level when it comes to the dynamic mode migrations of MCS.

A. Contributions

We address the interference problem on the shared memory bus in multi-core MCS by making the following contributions. 1) We expose strengths and inherent limitations of currently used arbiters in traditional single-criticality systems upon their applicability to MCS (Section V). 2) Hence, we introduce

CArb, an arbitration mechanism for controlling accesses to the shared memory bus in MCS (Section VI). CArb is a hierarchical two-tier arbiter that is, to our knowledge, the first to be criticality- and requirement-aware. This is necessary for two reasons. First, it results in optimal service allocation to tasks to meet their temporal requirements (Section VII). Second, it prioritizes tasks of higher criticality if the current set of memory requirements of all tasks is not schedulable. This is a vital characteristic when moving to higher modes in MCS (Section VIII). 3) We illustrate a methodology to decompose worst-case (WC) memory access latencies from the WC computation latencies experienced by a task. This has the advantage of allowing various MCS scheduling policies on cores to co-exist and operate in tandem with CArb; thereby, not imposing any restrictions on processor scheduling. 3) We propose two mechanisms to dynamically adapt the memory bus arbitration at run time to respond to increases in the monitored execution times of tasks. We show how these mechanisms can mitigate these increases; thus, in some cases, postpone or even eliminate the system need to switch to a degraded mode. We believe that avoiding these switches is highly desirable because of their notoriously huge overheads. In addition, the proposed mechanisms prevent unnecessary suspension of lower-critical tasks. 4) We experiment with a case-study from the avionics domain as well as with synthetic experiments. Our results show that CArb is well-suited for bus arbitration in multi-core MCS.

II. Related Work

A. Scheduling Techniques

There are several research efforts that investigate scheduling tasks with mixed criticalities on the same platform [3]–[7]. While earlier works primarily focus on single-core platforms [3], [4], recent efforts propose strategies for deploying MCS onto multi-core platforms [5]–[7]. An on-going survey [17] maintains a comprehensive list of these efforts. There exist two practical issues in these efforts that are related to our proposal. 1) They suspend LO-critical tasks at the HI-mode [17]; thus, having no guarantees for the those tasks that are deemed low criticality, but still critical to some degree. Hence, this suspension can result in some safety issues [2]. 2) They do not address temporal interferences between tasks arising from accessing resources that are shared amongst cores such as memory buses, caches and DRAMs. For the first issue, we promote fine-grained rescheduling to allow higher-critical tasks to meet their new requirements, while not suspending the lower-critical ones, if possible (Section VIII). To our knowledge, [9], [16] are the only existing efforts to address the second issue.

The approach in [16] employs a software-based throttling mechanism to manage accesses to the shared main memory. It assigns a memory access budget to each core, and when a non-critical core exceeds its budget, [16] throttles it to guarantee requirements of the critical core. We find that this approach is suitable for only dual-criticality MCS, where each task is either critical or non-critical. For MCS with multiple criticalities, [16] faces the aforementioned issue of throttling lower-critical tasks. In addition, [16] mandates mapping all critical tasks to the same core. Two drawbacks arise from this requirement. 1) It limits the applicability of this technique to other scheduling approaches that do not meet this requirement. 2) Systems with large number of critical tasks cannot use this approach if critical tasks are not schedulable in a single core. The technique in [9] arbitrates amongst memory requests from all tasks using conventional RR and FCFS policies. However, these arbiters, as we discuss in Section V, are agnostic to the distinct criticality and requirements of tasks, as they allocate the same service to all tasks. As a consequence, the bounds obtained in [9] are pessimistic. We address these limitations by proposing CArb that arbitrates accesses to the shared memory bus according to both criticality and timing requirements of all tasks. Utilizing CArb, we illustrate how to distinctly allocate service to tasks in an optimal fashion.

B. Shared Resources amongst cores

Recent works address interference in MCS due to shared DRAMs such as [12] [13], and shared caches amongst cores such as [14] [15]. These works do not address the interference occurring on the memory bus connecting various cores with these shared memories. Our work is orthogonal to these efforts and can coherently operate with them to address the interference problem on all resources of MCS; thus, enabling safe co-existence of tasks with different CLs on the same multi-core platform.

III. System Model

Figure 1 depicts the system considered in this paper. We assume a multi-core system, where each core executes a single task. This task runs until completion on the dedicated core. We support any mapping of tasks to cores. This allows the integration of CArb with a wide variety of existing task mapping schemes. Existing cores share inter-core platform resources. Specifically, off-chip DRAM, on-chip last-level cache (LLC), and the memory bus connecting cores to the LLC. We assume that the interference on shared DRAM is resolved using existing techniques such as partitioning [12] or requirement-aware scheduling [13]. Similarly, interference on shared data in the LLC is addresses by deploying cache partitioning or colouring [14]. Accordingly, this work focuses on the interference problem on the shared memory bus, and its impact on the total execution time of various tasks. We consider a mixed criticality system with n criticality levels. We classify tasks according to their criticality into groups that we denote as *classes*. Hence, there exist a set of n classes. Each class is defined as $C_l = \langle L_l, \Gamma_l \rangle$, where L is the criticality level and Γ_l is the total number of tasks in C_l.

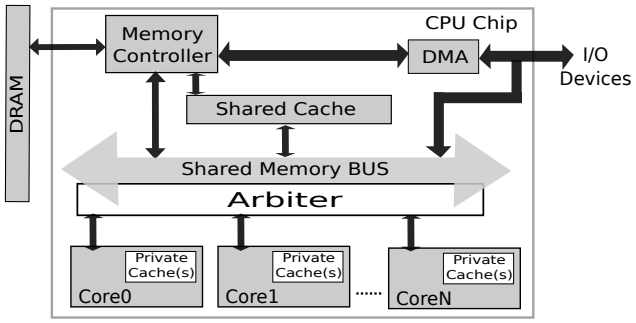


Fig. 1: Multi-core architecture.

Higher values of L denote higher criticality levels. A task is characterized as: $\tau_{jl} = \langle L_l, T_{jl}, D_{jl}, E_{jl}, S_{jl}(L), I_{jl}(L), \Lambda_{jl} \rangle$ where: T_{jl} is the minimum inter-arrival time of task jobs which represents the task period. D_{jl} is the task deadline, where $D_{jl} = T_{jl}$. S_{jl} is the WCET of any job of task τ_{jl} when τ_{jl} runs in isolation (no inter-task interference). I_{jl} is the WC additional latency due to inter-task interference. E_{jl} is the total WCET. Since the interference delays can be considered to be additive to the task's WCET in isolation [18], we assume that $E_{jl} = S_{jl} + I_{jl}$. Each task has an S value for each CL in the system. This is a primary characteristic of MCS. The intuition behind these different values per task is as follows. The computed WC times of a task are estimates calculated using extensive testing and/or static analysis methods. Hence, based on the accuracy and pessimism levels of these methods, different estimates may exist. The higher the criticality level is, the more pessimistic the values are [1], [8]. Λ_{jl} is the maximum number of memory accesses issued by any job of τ_{jl}. It is worth noting that CArb makes no assumption about the memory access rate of tasks. Λ_{jl} represents the WC number of memory accesses per period over all periods of the task, similar to [19]. This is analogous to the execution time of the task. Both the number of memory accesses, and the execution are different from one period to another; nonetheless, the task model only considers the WC execution time of a task. Both S and Λ can be collected using either measurement-based techniques or static analysis tools. For sake of simplicity, we assume that Λ is constant for all CLs. In section VII-C, we generalize the model to consider $\Lambda(L)$ as function of the CL. Since the number of running tasks varies with regard to the CL, so does the interference amongst these tasks. Hence, I is a function in the CL, $I(L)$. This task model considers two extensions to the standard MCS model [17]. 1) We assume an arbitrary number of CLs. We promote this for two reasons. First, to not limit the integration support of CArb to only dual-criticality scheduling mechanisms; rather, CArb supports also mechanisms with more criticality levels such as [20]. Second, we encourage MCS models that adopt more criticality levels because current industrial standards, for instance in avionics domain, call for up to five levels. Examples of these standards include IEC 61508, DO-178B, DO-254 and ISO 26262 [8]. 2) Decomposition of total execution time, E, into S and I. This enables memory bus arbitration to optimize service allocation to tasks according to their deadline requirements. A more detailed discussion on this decomposition is in the following section.

IV. Execution Time Decomposition

Migrating MCS onto multi-core platforms with inter-core shared resources, the interference delay due to shared resources amongst cores becomes an eminent component in the total WCET. Therefore, we claim that focusing on the interference delays of tasks is as necessary as the traditional focus on WCETs calculated in isolation. In consequence, we incorporate the total WCET E with its two components S and I in the proposed MCS model. This allows core scheduling techniques to focus on optimizations that affect S and shared memory arbitration techniques to minimize or eliminate I. In this paper, we employ this decomposition process, and show that such separation enables attaining optimal solutions to the interference problem. This section illustrates the decomposition process, while Section VII-B targets optimal allocations.

Existing approaches in scheduling MCS usually formulate the requirement on S relative to D as a schedulability condition. Tasks in the system are schedulable under the scheduling scheme only if they satisfy this condition. For the aforementioned reasons, we argue for substituting S with E in the schedulability analysis of multi-core MCS, and we assume S and I are additive such that $E = S + I$. Hence, if S is known beforehand, the schedulability condition turns into a requirement on the total interference delay encountered by each task such that the set of tasks is schedulable. Further, for static analysis purposes, the WC interference delay per memory request, M, must be assumed. Recall that the total number of memory requests of a task is Λ; thus, $I = M \times \Lambda$. Both S and Λ are predetermined task's characteristics. Subsequently, from the schedulability condition, we derive a requirement on M_{jl} for each τ_{jl} such that the set of tasks is schedulable. Since this paper focuses on the interference on shared memory buses, we denote this condition as *the memory latency requirement*. The arbiter must allocate services to tasks such that the maximum memory latency of any request does not violate this requirement.

A. Illustrative Example

We show how to derive the memory access requirement from the schedulability condition. We use the partitioning algorithm proposed by Sha [21] as an example of a scheduling policy used in the avionics domain. The policy in [21] splits the set of tasks into partitions. All tasks of a partition must execute on the same core. A class, as defined in Section III, can consist of multiple partitions. Hence, tasks of same class can execute in parallel on multiple cores if they belong to different partitions. In consequence, the partitioning algorithm of [21] resembles a general scheduling example by allowing tasks of same as well as different criticalities to run simultaneously; thus, interfere on shared memory. Under this algorithm, a time-division-multiplexing (TDM) scheduler assigns slots to partitions and a rate monotonic (RM) algorithm schedules tasks of the same partition. Sha [21] proved that the sufficient schedulability condition for each partition is:

$$U_r \leq \gamma_r \big((\frac{2}{2 - \hat{U}_r})^{\frac{1}{\gamma_r}} - 1 \big). \tag{1}$$

Where for partition r, γ_r is the total number of tasks in r, U_r is the total utilization of those tasks, and $\hat{U}_r$ is the partial utilization granted to r, which is the number of TDM slots granted to the partition divided by the total number of slots in the TDM schedule. From this schedulability condition for a partition, we compute the memory latency requirements using the following procedure:

(1) Given that the utilization of all tasks in partition r is $U_r = \sum_{j=1}^{k_r} \frac{S_{jr}}{D_{jr}}$ and substituting S with $E = S + I$, as discussed earlier, then:

$$\sum_{j=1}^{\gamma_r} \frac{S_{jr} + M_{jr} \times \Lambda_{jr}}{D_{jr}} \leq \gamma_r \big((\frac{2}{2 - \hat{U}_r})^{\frac{1}{\gamma_r}} - 1 \big)$$

(2) Recall that S_{jr} and Λ_{jr} are predetermined for all tasks, then the memory access latency requirements per task to satisfy

schedulability condition is obtained by Equation 2. Memory access latencies of all tasks of that partition must satisfy the condition in Equation 2. Notice that if a different scheduling algorithm is used, a similar procedure can be conducted to obtain the corresponding condition.

$$\sum_{j=1}^{\gamma_r} \frac{M_{jr} \times \Lambda_{jr}}{D_{jr}} \leq \gamma_r\big((\frac{2}{2-\hat{U}_r})^{\frac{1}{\gamma_r}} - 1\big) - \sum_{j=1}^{\gamma_r} \frac{S_{jr}}{D_{jr}} \qquad (2)$$

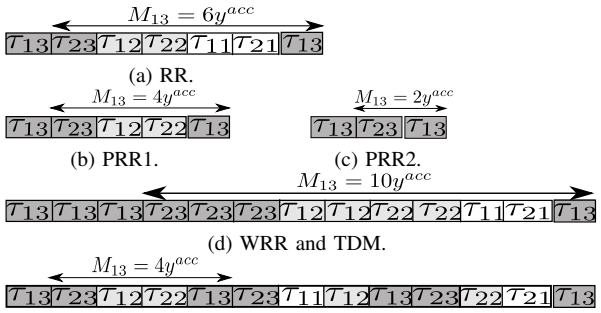
V. Applicability of real-time arbiters in MCS

We study commonly used arbiters in traditional real-time systems to investigate their applicability on MCS. Particularly, we focus on RR arbiters: bare RR, prioritized RR (PRR), weighted RR (WRR), and harmonic RR (HRR) in addition to TDM arbiters: contiguous TDM, and work-conserving distributed TDM. We argue that an adequate arbiter for MCS must posse two features: *requirement-awareness* and *criticality-awareness*. Requirement-awareness implies that the arbiter is able to allocate service to tasks based on their temporal requirements. Comparatively, criticality-awareness is achievable when the arbiter allocates service to tasks relative to their criticality. We evaluate each arbiter with regard to adopting these two features using Figure 2 for illustration. In Figure 2, we assume a system with three criticality levels and 6 tasks. $\mathcal{T}_{13}$ and $\mathcal{T}_{23}$ are the highest critical, $\mathcal{T}_{12}$ and $\mathcal{T}_{22}$ are of medium criticality, and $\mathcal{T}_{11}$ and $\mathcal{T}_{21}$ are non-critical. y^{acc} is the access latency to the shared memory.

Bare RR. RR is dynamic and simple to implement. The arbiter equivalently rotates amongst tasks (Figure 2a). The WC latency of a request from any task is bounded by the number of tasks in the system; hence, RR assures predictability. RR allocates the same service to all tasks regardless of their distinct criticality and timing requirements. For instance, in Figure 2a, all tasks encounter the same WC latency of $6y^{acc}$ cycles. Hence, RR is neither criticality-aware nor requirement-aware; thus, ill-suited to MCS.

PRR. Authors in [22] address the deficiencies of RR by proposing PRR. The arbiter conducts RR arbitration amongst critical tasks only. Non-critical tasks gain access only on *slack slots*, which are slots when there are no ready requests from any critical task. This solution targets systems with dual-criticality. Applying PRR in MCS with more than two levels, critical tasks (but not the most critical) can be scheduled by two approaches. 1) They share the schedule with the most critical tasks; hence, attain as much service as them even though they may have different requirements. In Figure 2b, both tasks of C_2 and C_3 have a WC latency of $4y^{acc}$. 2) They share the slack slots with non-critical tasks; thus, they have no timing guarantees, and may miss their deadlines (Figure 2c). Accordingly, we find that PRR's applicability is limited to dual-criticality systems where tasks with the lower CLs have no requirements.

WRR and Contiguous TDM. Unlike bare RR, WRR [23] is capable of allocating different amounts of service (slots or weights) to tasks based on their requirements as Figure 2d illustrates. Similar capability exists for contiguous TDM. The major difference between contiguous TDM and WRR is that TDM arbiters are, in general, non-work conserving. A slot assigned to a task will remain idle if there are no ready requests from this particular task even if there are ready requests from other tasks. On the other hand, WRR is work-conserving, and assigns idle slots to the first task with a ready request. Deploying either contiguous TDM or WRR, tasks with higher weights (or number of slots in TDM) encounter less average-case latency; though, the WC latency of requests from these tasks is either the same as or higher than bare RR. For example, in Figure 2d, the most critical task $\mathcal{T}_{13}$ obtains $1/4$ of the total slots. Notice that it suffers a WC latency of $10y^{acc}$ cycles compared to only $6y^{acc}$ cycles in bare RR. This is because in the WC, a request from any task (critical or non-critical) must wait for requests from all other tasks before it gets an access. Consequently, their deployment in MCS leads to pessimistic WCETs, and may not satisfy task requirements.



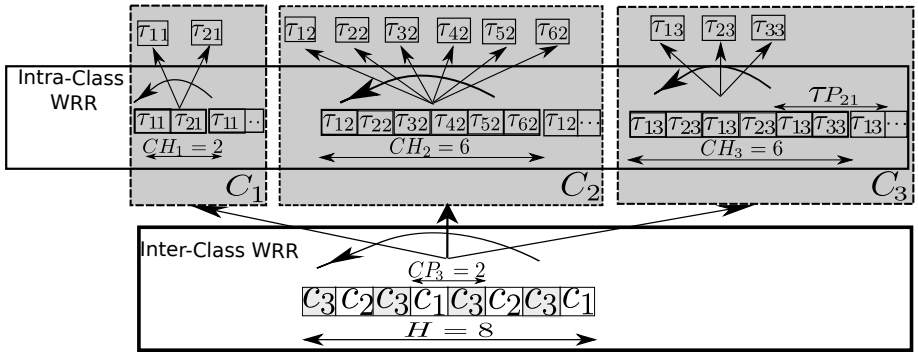
(a) RR. (b) PRR1. (c) PRR2. (d) WRR and TDM. (e) Harmonic WRR and distributed TDM.

Fig. 2: Real-time arbiters.

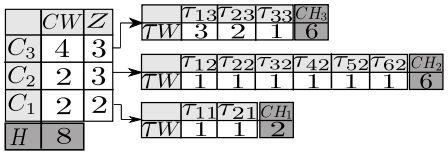
HRR and work-conserving distributed TDM. HRR [24] and work-conserving TDM [13] address this pessimism in WRR and contiguous TDM by evenly distributing slots assigned to tasks across the schedule as shown in Figure 2e. They have a different WC bound per task based on its requirements. Therefore, they are requirement-aware. Nevertheless, they assign service to tasks solely based on their timing requirements and not criticality. Upon applied to MCS, being non-criticality aware has two drawbacks. 1) In both approaches [13], [24], meeting the requirements for lower-criticality is as important as meeting those of the most-critical tasks. As aforementioned, in MCS, importance of fulfilling task requirements is relative to its criticality. For instance, in the automotive domain, it is crucial that the anti-lock brake system (ABS) meets its requirements over the proper functioning of the radio system does [25]. 2) Under the dynamic migration between various modes of the MCS system, a non criticality-aware approach is agnostic to which tasks must meet their requirements under all modes and which ones, on the other side, can be throttled at certain situations.

VI. CArb: Proposed Arbitration Scheme

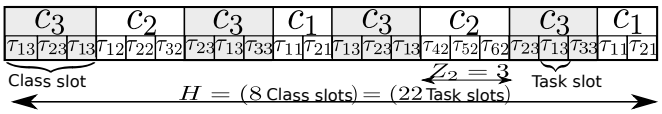
Motivated by the limitations of existing arbiters for MCS, we introduce CArb: a criticality- and requirement-aware arbiter that is configurable. CArb deploys a hierarchical two-tier arbitration scheme to manage accesses to the shared memory bus. It classifies tasks by their criticality grouping tasks of same CL in a class. Then, it executes a *harmonic WRR inter-class* arbitration among classes in the first tier, and a *harmonic WRR intra-class* arbitration amongst tasks of the same class in the second tier. Figure 3 depicts a system with 11 tasks classified into three classes C_1, C_2 and C_3, where C_3 is the most-critical. We use it as an example to illustrate CArb's operation.



(a) Inter- and intra-class arbitration.



(b) Look-up table required for schedule parameters.



(c) Final CArb schedule.

Fig. 3: Memory bus arbitration using CArb.

Inter-class Arbitration. CArb has two types of slots: *class slots* and *task slots*. A class slot consists of one or more task slots and is granted to a single class. The number of task slots in a class slot is generally distinct per class and is defined by its *window size*, Z_l. Since CArb deploys a harmonic WRR arbitration amongst classes, the number of class slots assigned to C_l is relative to its *class weight*, CW_l. *Schedule hyperperiod* is the summation of all class weights, $H = \sum_{l=1}^{n} CW_l$ such that CArb repeats the same schedule every H. Subject to CW_l and H, each class gets a slot every $CP_l = \frac{H}{CW_l}$, which we denote as CArb*'s class period*. In Figure 3, class weights CW_1, CW_2 and CW_3 are $2, 2$ and 4, respectively comprising a hyperperiod of $H = 8$, while the class window sizes Z_1, Z_2 and Z_3 are $2, 3$ and 3, respectively. Algorithm 1 describes the inter-class arbitration mechanism. For each class slot, a flag bit is reset to indicate that the slot is not allocated yet (line 3). Then, the arbiter iterates through the set of classes starting from the most critical one. For each class, C_l, the arbiter checks if C_l has to start a new period. If C_l starts a new period, a flag bit, denoted as *class grant* CG_l, is reset (line 6). If $CG_l = 0$, which implies that C_l is ready to be scheduled, and the current class slot is not allocated yet, CArb allocates this slot to C_l (lines 7 to 11). At this step, CArb moves to the intra-class arbitration to schedule tasks of C_l. Afterwards, CArb switches to the next class slot and repeats the same process again (the loop in lines 2 to 13) for H slots, then starts a new hyperperiod with same schedule. The inter-class WRR shown in Figure 3a exemplifies a schedule resulting from Algorithm 1, where the schedule repeats every 8 class slots.

Intra-class Arbitration. Recall that the inter-class tier grants Z_l task slots to C_l every class slot assigned to it. This results in a total of $CW_l \times Z_l$ task slots every H. The role of the intra-class arbitration is to distribute these task slots amongst tasks of C_l to satisfy their requirements. This is achieved by executing a per-class schedule that deploys a harmonic WRR arbitration amongst tasks of the same class. Thus, the *task weight*, $\mathcal{T}W_{jl}$, determines the number of task slots assigned to $\mathcal{T}_{jl}$. Summation of all task weights constructs the *class*

Algorithm 1: CArb(...) – inter-class arbitration.

```
   Input: CW_l, Z_l, Γ_l ∀l in [1, n]
1  H ← SUM_∀l(CW_l);
2  foreach (classSlot in [0, H − 1]) do
3      allocated ← false;
4      foreach (l in [n, 1]) do
5          CP_l ← H/CW_l;
6          if (mod(classSlot, CP_l) = 0) then CG_l ← 0;
7          if (CG_l = 0 and allocated = false) then
8              CG_l ← 1;
9              scheduleClass(Z_l, Γ_l);
10             allocated ← true;
11         end
12     end
13 end
```

Algorithm 2: scheduleClass(...) – intra-class arbitration.

```
   Input: Z_l, Γ_l
1        τW_jl ∀j in [1, Γ_l]
2  CH_l ← SUM_∀l τW_jl;
3  foreach (taskSlot in [0, Z_l − 1]) do
4      allocated ← false;
5      foreach (j in [1, Γ_l]) do
6          τP_jl ← CH_l/τW_jl;
7          if (mod(indx_l, τP_jl) = 0) then τG_jl ← 0;
8          if (τG_jl = 0 and allocated = false) then
9              τG_jl ← 1;
10             inc(indx_l);
11             if (τ_jl has a waiting request) then
12                 scheduleTask(τ_jl);
13                 allocated ← true;
14             end
15         end
16     end
17 end
```

hyperperiod: $CH_l = \sum_{j=1}^{K_l} \mathcal{T}W_{jl}$. The intra-class arbitration repeats the same task schedule amongst tasks of C_l every CH_l, while $\mathcal{T}_{jl}$ gets $\mathcal{T}W_{jl}$ task slots every CH_l.

$\mathcal{T}P_{jl} = \frac{CH_l}{\mathcal{T}W_{jl}}$ is the CArb*'s task period* such that CArb must grant a task slot to $\mathcal{T}_{jl}$ every $\mathcal{T}P_{jl}$ task slots in CH_l. In Figure 3, weights of C_3's tasks $\mathcal{T}W_{13}$, $\mathcal{T}W_{23}$ and $\mathcal{T}W_{33}$ are $3, 2$ and 1, respectively. This results in a class hyperperiod of $CH_3 = 6$. Algorithm 2 illustrates the intra-class arbitration process. Clearly, it is very similar to the first tier arbitration amongst classes with some conceptual differences. The intra-class arbitration executes a distinct schedule per class— see for example the task schedules if C_1, C_2 and C_3 at the intra-class tier in Figure 3a. CArb tracks the number of task slots allocated to C_l in the current schedule hyperperiod by the counter $indx_l$ (line 7). For the current task slot, with particular $indx_l$ value, it checks if $\mathcal{T}_{jl}$ has to start a new period (line 4). It repeats this check for all tasks in C_l. $indx_l$ is reset at the start of every H. We dictate the task slot width to allow for one access to the shared memory finish. Hence, once CArb grants access to a request from any task, it cannot be preempted. This is mandatory to guarantee predictability, while keeping the arbiter feasibly simple to implement. Having CArb implementing WRR, it is a work-conserving arbiter. For any task slot, if a task does have a ready request, CArb will allocate this slot to the first task in the schedule with a ready request (line 8 in Algorithm 2). The final schedule that CArb implements by executing both tiers of arbitration is akin to the instance shown in Figure 3c.

Area Overhead. For CArb to be able to execute a bus schedule satisfying memory requirements, it seeks the pre-knowledge of the variables that comprises this schedule, which we denote as *schedule parameters*. Particularly, it requires the values of $\mathcal{T}W_{jl}$, CW_l and Z_l for all classes and tasks. We formulate an optimization problem in Section VII to specify the optimal values of these variables and solve this problem offline based on requirements obtained in Section IV. Hence, obtained schedule parameters are stored in a configurable look-up table during boot time. For the example schedule shown in Figure 3, we illustrate the look-up table structure in Figure 3b. Let each schedule parameter require a $32bits$ (or $4B$) register. Generally, for n classes and Γ_l tasks per class, class parameters (CW_l and Z_l) demand $(8 \times n)B$, while task parameters per class require $(4 \times \Gamma_l)B$. Accordingly, a total storage of $440B$ is sufficient to store the schedule of a system with 5 classes (5 is the maximum number of criticality levels specified by standards) and 100 tasks per class. We believe that this is a negligible area overhead for commodity multi-core systems.

VII. WC Analysis and Problem Formulation

When using CArb, a request to the shared memory bus incurs two types of latencies, scheduling latency and access latency. Definitions 1 and 2 formally define these latencies.

Definition 1: **Scheduling latency**, $y^{sch}_{jl,r}$, of a request req_r generated by $\mathcal{T}_{jl}$ is measured from the time stamp of its issuance until it is granted access to the memory bus. $y^{sch}_{jl,r}$ is due to requests from other tasks scheduled before $\mathcal{T}_{jl}$.

Definition 2: **Access latency** is the latency suffered by a request generated by $\mathcal{T}_{jl}$ while it is accessing the shared memory. We assume that accessing the shared memory takes a fixed latency, y^{acc}. This latency can be considered as the WC access latency of the shared memory. Determining the value of y^{acc} is outside the scope of this paper and existing work can be used to determine it both for LLCs [14] and DRAMs [13].

A. WC analysis

Lemma 1: The total WC latency of a memory request generated by $\mathcal{T}_{jl}$, denoted as y^{tot}_{jl}, is computed as follows.

$$y^{tot}_{jl} = \Bigg(\Big(\sum_{v=1, v\neq l}^{v=k_l} \left\lceil \frac{\mathcal{T}W_{vl}}{\mathcal{T}W_{jl}} \right\rceil \Big) + \Bigg(\left\lceil \frac{\mathcal{T}P_{jl}}{Z_l} \right\rceil \times \sum_{\substack{\forall e | (e \neq l \wedge \\ f_e \in \chi_l)}} \Big(\left\lceil \frac{CW_e}{CW_l} \right\rceil \times Z_e \Big) \Bigg) + 1 \Bigg) \times y^{acc}$$

Proof: The WC scheduling latency occurs when a request waits for the WC number of requests before it can get access to the resource. Recall that scheduling latency suffered by a request from $\mathcal{T}_{jl}$ is due to requests from other tasks scheduled before $\mathcal{T}_{jl}$. These tasks belong either to the same class and cause *intra-class scheduling latency* or other classes and cause *inter-class scheduling latency*.

WC intra-class scheduling latency. In WC, during $\mathcal{T}P_{jl}$, there are $\left\lceil \frac{\mathcal{T}W_{vl}}{\mathcal{T}W_{jl}} \right\rceil$ slots assigned to $\mathcal{T}_{vl}$ ($v \neq l$). Accordingly, during $\mathcal{T}P_{jl}$, the WC number of scheduled requests from tasks belonging to the same class is: $\sum_{v=1}^{v=k_l} \left\lceil \frac{\mathcal{T}W_{vl}}{\mathcal{T}W_{jl}} \right\rceil$.

In Figure 3c, a request from $\mathcal{T}_{33}$ has to wait for 3 requests from $\mathcal{T}_{13}$ and 3 requests from $\mathcal{T}_{23}$.

WC inter-class scheduling latency. In WC, C_l has to wait for $\mathsf{MIN}(CP_l, n)$ distinct classes before it is granted a class slot, where each of these classes is assigned $\left\lceil \frac{CW_e}{CW_l} \right\rceil$ class slots. Furthermore, these classes are assigned the maximum number of task slots. Let $f_e = CW_e \times Z_e$ be the maximum number of task slots. In Figure 3c, $f_1 = 4, f_2 = 6$ and $f_3 = 12$. Equation 3 calculates the WC number of task slots CArb grants to other classed before it grants C_l a class slot. $\chi_l = \mathsf{MAX}(F, \mathsf{MIN}(CP_l, n))$ represents the largest $\mathsf{MIN}(CP_l, n)$ elements of F where $F = \{f_1, f_2, ..., f_n\}$. χ_l identifies the $\mathsf{MIN}(CP_l, n)$ classes with maximum number of task slots to represent the worst-case for C_l.

$$\sum_{\substack{\forall e | e \neq l \wedge \\ f_e \in \chi_l}} \Big(\left\lceil \frac{CW_e}{CW_l} \right\rceil \times Z_e \Big) \tag{3}$$

In addition, having C_l attained a class slot, does not necessarily imply that $\mathcal{T}_{jl}$ attains a task slot. Recall that once C_l attains a class slot, Z_l task slots are granted to its tasks, and $\mathcal{T}_{jl}$ gets a task slot every $\mathcal{T}P_{jl}$ at the C_l's schedule. Hence it gets one task slot every $\left\lceil \frac{\mathcal{T}P_{jl}}{Z_l} \right\rceil$ class slots granted to C_l. Consequently, a request from $\mathcal{T}_{jl}$ suffers from a WC inter-class scheduling latency of $\left\lceil \frac{\mathcal{T}P_{jl}}{Z_l} \right\rceil \times \sum_{\substack{\forall e | e \neq l \wedge \\ f_e \in \chi_l}} \Big(\left\lceil \frac{CW_e}{CW_l} \right\rceil \times Z_e \Big)$. In Figure 3c, a request from $\mathcal{T}_{33}$ has to wait in WC for $\Big(\left\lceil \frac{CW_2}{CW_3} \right\rceil \times Z_2 \Big) \times$

$\left\lceil \frac{\tau P_{33}}{Z_3} \right\rceil = (1 \times 3) \times 2 = 6$ task slot granted to other classes before it is granted an access. Finally, we add 1 to account for the access latency of the request itself and multiply by y^{acc} to transform slots into cycles. In conclusion, the total WC latency of any request is equal to the value computed in Lemma 1. ■

B. Optimization problem formulation

Target Function. We formulate the schedule construction process as an optimization problem. The target is to generate the harmonic schedule with minimum hyperperiod that satisfies requirements of all tasks. Hence, the schedule is optimal amongst the set of harmonic schedules. Note that there may exist a non-harmonic schedule with a shorter schedule hyperperiod, which can be obtained using either unconstrained search or heuristic solutions (see for example [26]) than CArb. Since CArb is a hardware arbiter, we consider the harmonic property to minimize the area overhead as discussed in Section VI, while allowing for 100% bus utilization. We determine the optimal values of weights and window sizes assigned to classes and weights assigned to tasks to construct that schedule. Therefore, we express the target function as:

$$\mathsf{MIN}(\sum_{l=1}^{l=n} CW_l \times Z_l).$$

Variables. The outcomes of this optimization problem are the task weights $\tau W_{jl,}$ class weights CW_l, and class window sizes Z_l that construct the schedule. Accordingly, using these values, task periods $\tau P_{jl,}$ class periods CP_l, class hyperperiods CH_l, and schedule hyper periods H are calculated.

Constraints. 1) The total WC latency must satisfy the memory access requirement obtained by Equation 2:

$$y_{jl}^{tot} \leq M_{jl} \tag{C.1}$$

2) Constraint C.2 prevents starvation at the inter-class arbitration tier. The lower bound, $CP_l \geq 2$, prohibits each class from starving other classes. If $CP_l = 1$, C_l will saturate the memory bus. The upper bound $CP_l \leq H$ prevents starving C_l as it assures that C_l will get at least one class slot in the schedule hyperperiod.

$$H \geq CP_l \geq 2 \tag{C.2}$$

3) Similarly, Constraint C.3 prohibits starvation at the intra-class arbitration tier:

$$CH_l \geq TP_{jl} \geq 2 \tag{C.3}$$

4) Three conditions are required to assert the periodicity characteristic such that CArb executes the schedule every H class slots or, equivalently every $CW_l \times Z_l$ task slots. First, the schedule hyperperiod, H, must be an integer multiple of CArb's class periods:

$$\frac{H}{CP_l} \in \mathbb{Z}_{>0}. \tag{C.4}$$

Second, every class hyperperiod, CH_l must be an integer multiple of CArb's task periods:

$$\frac{CH_l}{\tau P_{jl}} \in \mathbb{Z}_{>0} \quad \forall l \in [1, n] \forall j \in [1, \Gamma_l]. \tag{C.5}$$

Third, the total number of task slots granted to a class every H must be an integer multiple of the total number of required slots by tasks in that class:

$$\frac{CW_l \times Z_l}{CH_l} \in \mathbb{Z}_{>0}. \tag{C.6}$$

A final remark here is regarding constraint C.1. Recall that the condition on M_{jl} in Equation 2 depends on the value of S_{jl}, which is distinct per system mode. Two approaches can be followed based on two cases of the system requirements. 1) The first case is when the values of S for all tasks increase by the same ratio when system moves to higher levels. In this case, CArb stores only one optimal schedule that considers the value of S_{jl} for the lowest mode. Upon switching to higher modes, the operating system suspends lower-criticality tasks. Hence, their interference effect over all other tasks is eliminated. Since S increases by the same ratio for all tasks running at the new level, their schedule weights remain the same. Hence, the resulting schedule is sufficient to meet the requirements of the running tasks. 2) The second case is when the increase ratios of S are not the same among tasks. In this case, CArb requires a schedule per each mode l that corresponds to $S_{jl}(l)$. Since current standards acquires 5 levels, the total area overhead of these schedules is approximately $2KBs$, which we find acceptable for commodity systems.

C. $\Lambda(L)$: The WC number of memory accesses as a function of CL

So far, we have considered Λ_{jl} to be fixed for all CLs. However, since Λ and S are calculated using same methods, either analysis or measurements, the level of assurance on Λ can, akin to S, depend on CL. Hence, the higher the criticality level, the larger the value of Λ for the same task. To address this situation, for each l-mode, we run the optimization framework considering $S_{jl'}(l)$ and $\Lambda_{jl'}(l)$ for all tasks at that level. Namely, tasks of $l' \geq l$ since tasks of $l'' < l$ are already suspended by the system at l-mode. A resulting schedule per mode needs to be stored at the boot time. Upon mode switching, CArb switches to the corresponding optimal schedule to fulfil the new requirements of all tasks executing at that mode.

VIII. Dynamic Re-arbitration

A. Motivation

Suppose that the system operates at l-mode. Let the execution time of τ_{jl+1} with criticality $l+1$, s_{jl+1}, exceed its WCET value, $s_{jl+1} > S_{jl+1}(l)$. Then, the conventional approach is to switch the system to $(l+1)$-mode suspending all tasks of l criticality. Tasks of criticality $l'' < l$ are already suspended at l-mode. This approach creates two challenges that motivate our proposed *fine-grained rescheduling at the arbiter level*. 1) Suspending l-critical tasks at the $(l+1)$-mode entails having no guarantees for those tasks. 2) Due to high overheads upon mode switching at the system scheduling mechanism as studied by [27], minimizing those switches is highly desirable.

B. Proposed Solutions

Leveraging CArb, we can conduct a set of fine-grained rescheduling techniques at the arbiter hardware that can mitigate the aforestated two issues of the conventional approach. We illustrate two of these techniques.

***Scheme1: Prioritized* CArb.** This technique does not directly suspend tasks of l criticality. Instead, CArb allows them to access the shared memory bus only on slack slots when there are no ready requests from any task of higher criticality. As a consequence, this technique eliminates the interference from l-critical tasks. Thus, the interference suffered by $\mathcal{T}_{jl+1}$, i_{jl+1} decreases. Since the total execution time is $e_{jl+1} = s_{jl+1} + i_{jl+1}$, if the decrease in i_{jl+1} mitigates the observed increase in s_{jl+1} such that $e_{jl+1} \leq E_{jl+1}(l)$, there is no need to switch the mode. Otherwise, a mode switch is unavoidable. Since CArb schedule is statically predetermined, the maximum increase in the execution time of $\mathcal{T}_{jl+1}$ that this technique can mitigate before switching the mode, denoted as s^{max}_{jl+1}, is known offline for all tasks. During running time, the operating system monitors the execution time of all tasks and makes the following decisions. Decisions are shown for $\mathcal{T}_{jl+1}$ at l-mode; however, they hold for all tasks at all modes:

$$\begin{array}{lcl} s_{jl+1} \leq S_{jl+1}(l) & \rightarrow & \text{normal } l\text{-mode} \\ S_{jl+1}(l) < s_{jl+1} \leq s^{max}_{jl+1}(l) & \rightarrow & \text{apply prioritized CArb} \\ s_{jl+1} > s^{max}_{jl+1}(l) & \rightarrow & \text{switch to } (l+1)\text{-mode} \end{array}$$

Although prioritized CArb may appear similar to other priority-based arbiters, there are important differences. For instance, static-priority arbiters such as the one deployed in [12], does not provide any guarantees except for the highest-criticality tasks. In worst-case, tasks with the highest-criticality can issue requests forever; thus, starving other lower-criticality tasks. In other words, simple static-priority arbiters allow all tasks other than the highest-critical ones to issue requests only on slack time. In contrast, CArb applies prioritization only when a potential mode switch is discovered that CArb can avoid using re-arbitration. Amongst running tasks, only tasks of lowest-criticality (l-critical tasks at l-mode) issue requests on slack time. For example, assume a MCS with 5 criticality levels, where tasks are running at 1-mode, and the system monitors an increase in the execution time that CArb can mitigate without a mode switch. Accordingly, prioritized CArb forces tasks with 1-criticality to issue requests only on slack slots, and reallocate their slots to other tasks. All tasks of criticalities 2 to 5 are guaranteed to meet their requirements. On the other hand, if the system implements the aforementioned static-priority arbitration, only tasks of 5-criticality meet their requirements. There exist other static-priority arbiters that avoid starvation of lower-criticality tasks by deploying budgeting mechanisms such as the credit-control static priority arbiter (CCSP) [28]. However, they have shortcomings when applied to MCS. For example, during each period, lower-criticality tasks have to wait for all higher-criticality tasks to finish their budgets before it can issue a single request. Accordingly, they may or may not meet their temporal requirements. This has the same disadvantage as the contiguous TDM discussed in Section V. Contrarily, prioritized CArb distributes slots amongst running tasks in a harmonic way that is requirement-aware. This is true for all running tasks except for the lowest-criticality that CArb executes on slack time. In addition, once the monitored execution times decrease below their corresponding worst-case estimates, CArb reloads the normal schedule in the next schedule hyperperiod. Consequently, all tasks are guaranteed to meet their requirements.

Scheme2: Having an optimal schedule per mode. Prioritized CArb can be considered as a special case schedule, where weights of lower criticality tasks are set to 0. Although it successfully delays the mode switching, it can be considered the most conservative solution. Generally, based on the amount of increase in the execution times, there exist a set of possible CArb solutions that can offset this increase. On l-mode, each solution comprises an optimal schedule that satisfy the new requirements, with larger $S_{jl'}$ values for all $l' > l$, while it maximizes the allocated slots to tasks of l instead of setting their allocated slots to 0. For the task under analysis, $\mathcal{T}_{jl+1}$, since the set of real execution times between S_{jl+1} and s^{max}_{jl+1} is uncountably infinite, some execution time values must be selected to find the optimal schedule for. In addition, the larger the selected execution times are, the less the allocated service to l-critical tasks. Hence, a trade-off exists between the allocated service to tasks of l, and the area required to store the selected number of corresponding schedules. For instance, suppose that only one additional schedule is stored for each level that corresponds to the middle point of value $s^{sch2}_{jl+1} = \frac{s^{max}_{jl+1} + S_{jl+1}}{2}$; hence, for l-mode and task $\mathcal{T}_{jl+1}$, the operating system decisions become as follows:

$$\begin{array}{lcl} s_{jl+1} \leq S_{jl+1}(l) & \rightarrow & \text{normal } l\text{-mode} \\ S_{jl+1}(l) < s_{jl+1} \leq s^{sch2}_{jl+1} & \rightarrow & \text{apply optimal schedule2} \\ s^{sch2}_{jl+1} < s_{jl+1} \leq s^{max}_{jl+1}(l) & \rightarrow & \text{apply prioritized CArb} \\ s_{jl+1} > s^{max}_{jl+1}(l) & \rightarrow & \text{switch to } (l+1)\text{-mode} \end{array}$$

To obtain these additional optimal schedules for l-mode, Constraints C.1, and C.4 to C.6 apply only for $l' > l$ and the target function of the optimization problem changes to $\mathsf{MAX}(CW_l \times Z_l)$ only for tasks of l criticality.

Finally, there are important observations to highlight. First, suppose that CArb is executing scheme1 or scheme2, if all monitored execution times decrease below their WCET values, CArb can move back to the original optimal schedule. Second, the overheads of proposed schemes are negligible compared to mode switches as 1) they are conducted at the arbiter hardware which is much faster than processor rescheduling at the system level, and 2) the operating system does not require to handle any of the complex procedures of mode switching; instead, it just sends a signal to CArb to move to one of these schemes. At the end of each hyperperiod, CArb monitors whether it receives this signal from the system. In case of signal reception, CArb applies the appropriate re-arbitration at the next schedule hyperperiod.

C. Effect of Re-arbitration on Lower-criticality Tasks

Under normal operation, a MCS should satisfy requirements of both higher and lower criticality tasks. However, when the execution time of tasks increase, this may not be possible. The objective of the dynamic re-arbitration is to achieve the following two goals at each criticality level:

1) Guarantee the timing requirements of higher criticality tasks.

Use-case requirements					Processor Scheduling using [21]			Optimal CArb parameters	
τ_{jl}	D_{jl} (ms)	S_{jl} (ms)	Λ_{jl}	Partition	Core	U	memory access requirements	τw_{jl}	(CW_l, Z_l)
τ_{14}	25	1.06	500	1	1	0.25	$M_{14} \leq 5.02\mu s$	6	$(3,4)$
τ_{24}	50	3.09	500	2	1	0.25	$M_{24} \leq 8.11\mu s$	3	
τ_{34}	100	2.7	500	3	1	0.25	$M_{34} \leq 23.17\mu s$	1	
τ_{44}	200	1.09	500	4	1	0.25	$M_{44} \leq 45.96\mu s$	2	
τ_{13}	25	0.94	1000	5	2	0.4	$2M_{13} + M_{23} + M_{33} \leq 6.45\mu s$	6	$(6,4)$
τ_{23}	50	1.57	1000					4	
τ_{33}	50	1.68	1000					4	
τ_{43}	50	4.5	1000	6	2	3/5	$4M_{43} + 4M_{53} + 2M_{63} + M_{73} \leq 35.28\mu s$	3	
τ_{53}	50	2.94	1000					3	
τ_{63}	100	1.41	1000					3	
τ_{73}	200	6.75	1000					1	
τ_{12}	50	5.4	4000	7	3	0.4	$M_{12} \leq 1.77\mu s$	1	$(3,1)$
τ_{11}	50	2.4	2000	8	3	0.6	$4M_{11} + M_{21} + 4M_{31} + M_{41} \leq 28.77$	5	$(12,5)$
τ_{21}	200	0.94	2000					2	
τ_{31}	50	1.06	2000					5	
τ_{41}	200	2.28	2000					3	
τ_{51}	25	4.75	3000	9	4	1	$8M_{51} + 2M_{61} + M_{71} + 2M_{81} + 4M_{91} \leq 24.17$	20	
τ_{61}	100	12.87	3000					6	
τ_{71}	200	0.47	3000					3	
τ_{81}	100	1.24	3000					6	
τ_{91}	50	1.62	3000					10	

TABLE I: Experiment using the avionics use-case from Honeywell [1].

2) Provide lower criticality tasks with the maximum possible service after satisfying the first goal.

These two goals do not guarantee satisfying the timing requirements of lower-criticality tasks upon re-arbitration. In fact, upon re-arbitration, CArb will reduce the service delivered to lower-criticality tasks to satisfy requirements of higher ones. Nonetheless, the proposed fine-grained re-arbitration, unlike the traditional mode-switching approach, does not completely suspend lower-critical tasks unless needed. This is important since lower-criticality tasks are usually soft-real time tasks, which care about average-case rate of service or memory bandwidth. Hence, degrading their service is potentially a more practical solution than completely suspending them [2].

IX. Experimental Evaluation

We experimentally prototype CArb using a multi-core architectural simulator called MacSim [29] with CArb managing accesses to a shared L3 cache. We use a multi-core architecture of four x86 cores, private 16KB L1 and 256KB L2 caches per core, and a single 1MB L3 cache shared and partitioned amongst cores and operates at 1GHz. The access latency of the L3 cache is 50 cycles. The evaluation consists of three parts. In the first part, we evaluate CArb using a real use-case MCS requirements from the avionics domain. In the second part, we highlight the trade-offs associated with adapting CArb's schedule parameters. In the last part, we study the effectiveness of the proposed re-arbitration schemes.

A. Avionics Use-case

Experiment setup. We simulate a workload of 21 tasks derived from partition-based avionics system from Honeywell [1]. Table I tabulates for each task: the deadline, D_{jl}, WC execution time in isolation, S_{jl}, and the maximum number of memory access issued in a period, Λ_{jl}. The workload has 9 partitions (column 5 in Table I) and 4 criticality classes, C_1 to C_4. Since the actual task implementations are not publicly available, we implement in-house workloads that match requirements of these tasks. We deploy the algorithm proposed by Sha [21] to schedule tasks on cores using core assignments and utilizations given in columns 6 and 7 of Table I, respectively.

Obtaining CArb ***parameters.*** We use the schedulability condition in Equation 2 to construct the memory latency, which we show in column 8 of Table I. Afterwards, we implement the optimization framework proposed in Section VII in Matlab to obtain the optimal values of CArb's schedule parameters. According to the memory latency requirements, we get the optimal values for $\mathcal{T}W_{jl}$ (column 9 in Table I), CW_l and Z_l (column 10) that satisfy these requirements while minimizing the schedule hyperperiod.

Results. Figure 4 shows the WC latencies obtained when CArb executes the optimal schedule to arbitrate requests from all tasks to the shared L3 cache. Clearly, the values satisfy all the memory access latency requirements in column 8.

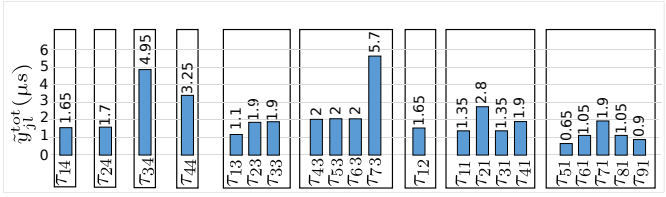


Fig. 4: Avionics use-case results.

B. Synthetic Experiments

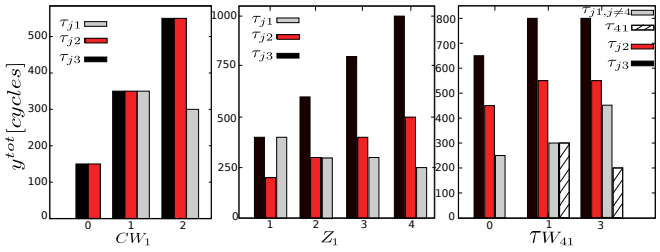
CArb is capable of meeting various system requirements by adapting its configurable parameters CW_l, Z_l and $\mathcal{T}W_{jl}$. Certainly, this adaptation involves a trade-off between requirements of different tasks. It is the role of the optimization framework to explore this trade-off and provide optimal values that satisfy all requirements. However, this experiment does not focus on discovering the optimal setting, but giving the reader a perspective on how parameters influence the outcome. In doing that, we disable the optimization framework and sweep each parameter to study its effect.

Experiment setup. We assume a system with three classes C_3, C_2 and C_1. C_3 has two tasks $\mathcal{T}_{13}$ and $\mathcal{T}_{13}$, C_2 has one task $\mathcal{T}_{12}$ while C_1 has four tasks $\mathcal{T}_{11}$ to $\mathcal{T}_{41}$. We run each task on a core and CArb manages accesses to a shared cache among cores. We conduct three experiments to: 1) vary CW_1, 2) vary Z_1, and 3) vary $\mathcal{T}W_{41}$. Table II shows the values of all parameters used in these experiments. Figure 5 delineates the results of each experiment.

Exp.	CW_3	CW_2	CW_1	Z_3	Z_2	Z_1	$\mathcal{T}W_{j3}$	$\mathcal{T}W_{j2}$	$\mathcal{T}W_{j1}$ $j \neq 4$	$\mathcal{T}W_{41}$
1	1	1	vary	2	1	4	1	1	1	1
2	2	2	4	1	1	vary	1	1	1	1
3	2	2	4	1	1	3	1	1	1	vary

TABLE II: Parameters of synthetic experiments.

Observations. 1) Increasing CW_1, CArb grants more class slots to C_1. Similarly, increasing Z_1 will increase the number of task slots assigned to C_1's tasks. Consequently, in both cases, y^{tot} of C_1's tasks will decrease at the expense of increasing y^{tot} of tasks in C_2 and C_3. We also show the amount of interference that C_1's tasks contribute to latencies of tasks belonging to other classes by illustrating the case when no task from C_1 is scheduled ($CW_1 = 0$). Since this situation implies starving C_1's tasks, the optimization framework prohibits it under normal conditions.
2) At certain values, increasing weights or window sizes of a class may not decrease y^{tot} of tasks in that class. For example, in Figure 5b, increasing Z_1 from 2 to 3 does not decrease y^{tot}_{j1}, while it has a negative effect on y^{tot} of tasks in C_2 and C_3. The rationale behind this observation is that increasing Z_1 from 2 to 3 do not, in fact, change the WC situation for tasks in C_1. According to the values of experiment 2 in Table II and $Z_1 = 2$, each task in C_1 attains 2 of 12 task slots in the schedule hyperperiod. Therefore, it has a WC scheduling latency of $\lceil \frac{12}{2} \rceil = 6$ slots. Increasing Z_1 from 2 to 3, each task in C_1 wins 3 of 16 task slots; hence, the WC scheduling latency becomes $\lceil \frac{16}{3} \rceil = 6$ slots. As a consequence, increasing Z_1 from 2 to 3 does not change y^{tot}_{j1}; however, it decreases average-case latency as tasks of C_1 execute more frequently.
3) By changing $\mathcal{T}W_{41}$ (Figure 5c), the intra-class schedule of C_1 changes. Apparently, increasing $\mathcal{T}W_{41}$ decreases y^{tot}_{41} at the expense of increasing y^{tot} of other tasks in C_1. Notice that with the exception of $\mathcal{T}W_{41}= 0$, changing $\mathcal{T}W_{41}$ has no effect on y^{tot} of tasks in C_2 and C_3. This is because the inter-class schedule remains the same. This is a consequence of the criticality-awareness of CArb as it separates class arbitration from task arbitration. Since assigning $\mathcal{T}W_{41}= 0$ will result in a free slot that will be assigned to other tasks, tasks of C_2 and C_3 have less y^{tot}.



(a) Class weight. (b) Class window size. (c) Task weight.

Fig. 5: Synthetic experiments (y-axis is the total WCL, y^{tot}).

C. Dynamic Re-arbitration

Experiment Setup. In this experiment, we investigate the capabilities of CArb's dynamic re-arbitration mechanisms proposed in Section VIII. We use the parameters in Table III to simulate a system with 3 classes. The partitioning algorithm in Sha [21] is used for core scheduling and Table III tabulates the used partitions and utilizations. According to the standard

τ	D (ms)	S (ms)	Partition	Core	U	Λ	$M(\mu s)$	τw	CW	Z
τ_{13}	5	1	1	1	0.5	2000	$M_{13} \leq 0.22$	1	4	1
τ_{23}	5	1	2	1	0.5	2000	$M_{23} \leq 0.22$	1		
τ_{12}	5	2	3	2	1	1000	$2M_{12} + M_{22} \leq 1.28$	1	2	1
τ_{22}	10	3				1000		1		
τ_{11}	10	2	4	3	1	2000	$3M_{11} + 2M_{21} \leq 2.9$	1	2	1
τ_{21}	15	8				2000		1		

TABLE III: Parameters of the dynamic case experiment.

MCS model, there are 3 modes of operations. 1-mode is the normal mode, where all tasks of all classes are operating according to the requirements given in Table III. In 2-mode, the operating system suspends tasks of C_1 and only tasks of classes C_2 and C_3 are utilizing the hardware. Finally, in 3-mode, tasks of C_1 and C_2 are suspended and only tasks of C_3 have the permit to execute. Normally, a switch from the 1-mode to the 2-mode occurs when the execution time of any task in C_2 or C_3 exceeds its S_{ij} value in Table III. To expose benefits of CArb dynamic re-arbitration, we postpone this mode switch and investigate if this re-arbitration is able to mitigate the increase in the execution time such that the requirements of tasks in C_2 and C_3 are met without suspending tasks of C_1. We model the increase in the execution time by decreasing the core operating frequency.

Observations. All tasks are affected by the frequency scaling. For clarity, we focus on results of C_2's tasks (partition 3). Figure 6 depicts $U_3 = \sum_{j=1}^{2} \frac{e_{j3}}{D_{j3}} = \sum_{j=1}^{2} \frac{s_{j3} + m_{j3} \times \Lambda_{j3}}{D_{j3}}$. The dotted line is the schedulability bound (right hand side of Equation 1).
1)Deploying CArb *without dynamic re-arbitration and disable mode switching*. As expected, decreasing the frequency, s_{12} and s_{22} increase and U_3 keeps increasing until violating schedulability condition (*noDynamic* plot in Figure 6).
2) Deploying CArb *with scheme1*. When s_{12} and s_{22} exceed their corresponding WCETs, S_{12} and S_{22}, CArb switches to the prioritized CArb mechanism, where tasks of C_1 gains access only on slack slots. As Figure 6 illustrates, *Scheme1* mitigates up to 12% and 18% increase percentages in s_{12} and s_{22}, respectively, without requiring the operating system to switch to mode 2. This results in postponing the mode switch from the frequency point of 990MHz to 950MHz. However, this comes at the expense of switching all tasks of C_1 to execute on slack slots.
3) Deploying CArb *with scheme*2. Given the trade-off discussed in Section VIII, we choose to store only one additional schedule configuration for scheme2 per mode. We choose a middle point between the WCETs and the maximum execution times that scheme1 can mitigate, where (s_{12}, s_{22})

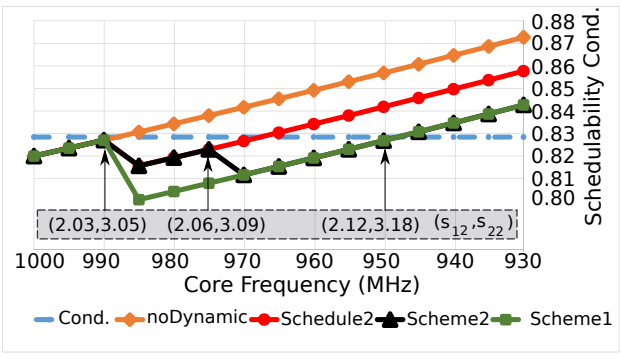


Fig. 6: Effect of decreasing core frequency on tasks of C_2.

equal $(2.06, 3.09)ms$, respectively, in Figure 6. This point is statically predetermined and we rerun the optimization framework to obtain the new optimal schedule. The obtained optimal schedule does not change the intra-class schedule and only reallocates the inter-class slots. This is achievable exclusively because CArb is criticality-aware with hierarchical scheduling. In the new schedule (*Schedule*2 in Figure 6), the class weights are the same as in Table III, while class window sizes change to $1, 2$ and 2 for C_1, C_2 and C_3 respectively. Instead of directly applying prioritized CArb once the execution times exceeds their WCETs, *Schedule*2 mitigates increases up to 6% and 9% in s_{12} and s_{22}, respectively. This occurs from 990MHz to 970MHz in Figure 6. In addition, it guarantees some service allocation on the memory bus to C_1's tasks. Afterwards, *Scheme*2 deploys the prioritized CArb (from 970MHz to 950MHz in Figure 6). Finally, a mode-switch is unavoidable when s_{12} and s_{22} exceed the point $(2.12, 3.18)ms$ (region after 950MHz in Figure 6).

X. Conclusion

We address the inter-task interference problem in multi-core mixed criticality systems by presenting CArb. CArb is a criticality- and requirement-aware bus arbiter adopting two-tier weighted round-robin arbitration. CArb has the following advantages: it does not restrict the scheduling policy for tasks on cores, and it supports any number of criticality levels. In addition, it optimally allocates service to tasks through configurable schedules loaded at boot time. CArb is capable to dynamically adapt its schedule under varying system conditions. This adaptation proves its effectiveness to mitigate the system need to switch to a degraded mode upon increases in the execution times of tasks. Finally, we evaluate CArb using avionics case-study and synthetic experiments.

References

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

[2] P. Graydon and I. Bate, "Safety assurance driven problem formulation for mixed-criticality scheduling," in *Workshop on Mixed-Criticality Systems*, 2013.

[3] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[4] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *IEEE Euromicro Conference onReal-Time Systems (ECRTS)*, 2008.

[5] R. M. Pathan, "Schedulability analysis of mixed-criticality systems on multiprocessors," in *IEEE Euromicro Conference on Real-time Systems (ECRTS)*, 2012.

[6] H. Li and S. Baruah, "Global mixed-criticality scheduling on multiprocessors," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[7] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin, "Mixed-criticality scheduling on multiprocessors," *Real-Time Systems*, 2014.

[8] I. Bate, A. Burns, and R. I. Davis, "A bailout protocol for mixed criticality systems," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[9] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in *IEEE International Conference on Embedded Software (EMSOFT)*, 2013.

[10] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *IEEE International Conference on Design, Automation and Test in Europe (DATE)*, 2010.

[11] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha, "Coscheduling of cpu and i/o transactions in cots-based embedded systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2008.

[12] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst, "A mixed critical memory controller using bank privatization and fixed priority scheduling," in *IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*.

[13] M. Hassan, H. Patel, and R. Pellizzoni, "A framework for scheduling dram memory accesses for multi-core mixed-time critical systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[14] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Outstanding paper award: Making shared caches more predictable on multicore platforms," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[15] N. Chetan Kumar, S. Vyas, R. K. Cytron, C. D. Gill, J. Zambreno, and P. H. Jones, "Cache design for mixed criticality real-time systems," in *IEEE International Conference on Computer Design (ICCD)*, 2014.

[16] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *IEEE Euromicro Conference on Real-Time System (ECRTS)*, 2012.

[17] A. Burns and R. Davis, "Mixed criticality systems: A review," *Department of Computer Science, University of York, Tech. Rep*, 2013.

[18] H. Yun, R. Pellizzon, and P. Valsan, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[19] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[20] P. Ekberg and W. Yi, "Schedulability analysis of a graph-based task model for mixed-criticality systems," *Real-Time Systems*, 2015.

[21] L. Sha, "Real-time virtual machines for avionics software porting and development," in *Real-Time and Embedded Computing Systems and Applications*. Springer, 2004.

[22] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware support for wcet analysis of hard real-time multicore systems," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 57–68.

[23] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, "Weighted round-robin cell multiplexing in a general-purpose atm switch chip," *IEEE Journal on Selected Areas in Communications*, 1991.

[24] M.-K. Yoon, J.-E. Kim, and L. Sha, "Optimizing tunable wcet with shared resource allocation and arbitration in hard real-time multicore systems," in *IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[25] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*. IEEE, 2010, pp. 13–22.

[26] A. Minaeva, P. Šcha, B. Akesson, and Z. Hanzálek, "Scalable and efficient configuration of time-division multiplexed resources," *Elsevier Journal of Systems and Software*, 2016.

[27] L. Sigrist, G. Giannopoulou, P. Huang, A. Gomez, and L. Thiele, "Mixed-criticality runtime mechanisms and evaluation on multicores," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[28] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-time scheduling using credit-controlled static-priority arbitration," in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

[29] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework user guide," *Georgia Institute of Technology*, 2012.