Maximum-Contention Control Unit (MCCU): Resource Access Count and Contention Time Enforcement

Jordi Cardona*†, Carles Hernandez*, Jaume Abella* and Francisco J. Cazorla*‡
* Barcelona Supercomputing Center † Universitat Politècnica de Catalunya ‡ IIIA-CSIC

Abstract—**In real-time systems, the techniques to derive bounds to the contention tasks can suffer in multicore build on resource quota monitoring and enforcement. Existing techniques track and bound the number of requests to hardware shared resources that each core (task) is allowed to perform. In this paper we show that current software-only solutions work well when there is a single resource and type of request to track and bound, but do not scale to the more general case of several shared resources that accept different request types, each with a different associated latency. To handle this (more general) case, we propose low-overhead hardware support called Maximum-Contention Control Unit (MCCU). The MCCU performs fine-grain tracking of different types of requests, preventing a core to cause more interference on its contenders than budgeted. In this process, the MCCU also helps verifying that individual requests duration does not exceed their theoretical bounds, hence dealing with scenarios in which requests can have an arbitrarily large duration.**

I. Introduction

Safety standards for critical real-time embedded systems (CRTES) in domains such as avionics and automotive require rigorous validation and verification (V&V). Timing V&V, the focus of this paper, provides evidence for the correct temporal schedulability of the system. This builds on deriving tight and reliable Worst-Case Execution Time (WCET) estimates. The quality of the WCET estimates often depends on engineer's previous experience. For instance, common industrial practice for timing analysis of complex platforms consists in running several tests measuring the highest execution or high watermark and adding an experience-based safety margin to it to cover the impact on 'unobserved' effects.

In CRTES, the advent of autonomous vehicles has increased the computing performance demands, that are satisfied by new high-performance hardware features [8], [22]. The other side of the coin is that high-performance resources like shared caches, interconnection networks and memory hamper timing V&V and in particular deriving tight and trustworthy timing WCET estimates [7], [13]. Without proper management of hardware shared resources, WCET on multicores can become much worse than their average behavior, defying the benefits of multicores in CRTES [23].

At hardware level, several designs [9], [10], [20], [23], [25] have been proposed to better factor in multicore contention in task's WCET estimates. However, to our knowledge, those solutions have not been fully adopted by industry yet for cost reasons. In particular, industry is reluctant to re-design and re-verify already-verified functional unit blocks (FUBs).

Software solutions for quota monitoring and enforcement (SQME) have been proposed to handle multicore contention in more generic processors with limited hardware support for time predictability [17], [24]. In general, SQME approaches build on limiting per task (core) maximum shared resources utilization. To that end, the operating system monitors task's activities using the available hardware performance monitoring counters (PMCs) and suspends tasks execution when their assigned budget is exhausted.

Contribution. We make the following two contributions to the field of multicore contention bounding in CRTES.

First, we identify key limitations of SQME. On the one hand, SQME work well only when are applied to one single shared resource in which all request types are assumed to have the same (worst) access latency. However, for the general case where several hardware shared resources accept different request types with different latencies, SQME generate unnecessary interrupts to determine whether tasks have consumed their budget, not only increasing task WCET but also hampering deriving bounds to WCET. On the other hand, SQME build on maximum per-request latency estimates that are derived empirically in real processors [16], but the lack of means to verify that those estimates actual bound maximum latencies which brings some uncertainty on the estimates built on top of those bounds. Moreover, in some AMBA bus implementations, a single request can potentially hold the bus for an unbounded duration. That means that setting quotas on request access counts, as done by SQME, does not suffice to bound contention time.

And second, to overcome the limitations of SQME, we propose to include in multicore processors a Maximum-Contention Control Unit (MCCU). The MCCU is a software-controllable low-overhead hardware unit able to ① accurately handle several resources dealing with several request types. Unlike SQME that may need to trigger several interrupts or perform frequent checks to handle utilization quotas, the MCCU only requires to trigger one interrupt when the quota allocated to a task is actually exhausted, allowing to significantly reduce the overheads of the enforcement mechanism. The MCCU, by monitoring requests duration, also ② allows monitoring seamlessly whether theoretical latency bounds derived empirically are effectively respected at all times and preserves quota enforcement even in the presence of requests with unbounded duration. The MCCU also ③ handles those (AMBA-compliant) scenarios in which a core/master can hold the bus for long time, preventing quota violations.

We show the effectiveness of the MCCU tailoring it for a 4-core multicore setup resembling the Cobham Gaisler NGMP processor for the space domain [4]. Our results for MediaBench show that while SQME can easily generate in the order of dozens of unnecessary interrupts to control quota on access counts, our proposed MCCU generates only interrupts when the task consumes all its contention quota removing any overheads and simplifying timing V&V.

In terms of implementation, the MCCU is an additional FUB connected as a slave to the AMBA interconnection network, thus not requires any modification on existing FUBs. In particular, the MCCU interface only needs some addressable space for being configured, snooping AMBA signals and raising specific interrupts whenever appropriate. This is arguably simpler than introducing small modifications in existing FUBs such as caches and memory controllers, which would require expensive and time-consuming re-verification costs.

II. Background and Related Works on Contention Control and Modelling for CRTES

Hardware techniques: Several hardware designs providing predictability-aware resource sharing exist [9], [10], [20], [23], [25]. These designs combine the use of time-predictable arbitration schemes and provisioning each core/task with separate queues in each hardware shared resource to avoid a given core to clog that resource. Unfortunately, those changes require the re-verification of affected FUBs such as cache memories, buses and memory controllers among others that is the main reason for chip vendors not having adopt these solutions yet.

Software techniques: Contention models build on a timing estimate of the isolation, i.e. without contention, execution time of the task τ_a, C_a^{isol}, to derive a multicore estimate of τ_a's execution time (C_a^{muc}). To that end, the models bound the contention τ_a's requests can suffer in the access to hardware shared resources, Δ_a^{cont} so that $C_a^{muc} = C_i^{isol} + \Delta_a^{cont}$.

Δ_a^{cont} is computed combining (i) the longest contention each of its request can suffer, L_{max} and (ii) the number of requests, n_a, performed by τ_a and its contenders running in the other cores, referred to as $c(\tau_a)$. Let $cr(x \rightarrow a)$ be the number of τ_a requests that contend in the access to a share resource with the requests of another concurrently running task t_x. It is defined as $cr_{b\rightarrow a} = min(n_a, n_b)$. Overall, the longest contention τ_a can suffer is defined as: $\Delta_a^{cont} = \sum_{\tau_x \in c(\tau_a)} cr_{x\rightarrow a} \times L_{max}$.

State of the art works on on this area can be classified into several groups. First those that derive bounds to the number of requests performed by tasks to shared resources [5], [17] (n) which is affected, among others, by tasks' input data and the execution paths they traverse.

Another strand of works derive bounds to the maximum contention delay each request of a task can suffer [7], [11], [16] (L_{max}). These works build on time-predictable arbitration policies (e.g. round-robin) used in many shared hardware resources in high-performance embedded processors. For instance, for round-robin the longest contention delay a request an suffer is $(Nc - 1) \times L$ when Nc is the number of requestors and L the duration of a request [18]. These works also build on documentation from processor manuals, when available, and carry strong validation through an extensive set of measurements to reduce the uncertainty on the validity of the bound [7], [16], which however cannot be removed.

And third, SQME works build on those bounds to derive contention models to bound Δ_a^{cont} in on-chip shared resources (e.g. bus and caches) [6], [11], [17] and enforce bounds (quotas) to task' access counts hence limiting Δ_a^{cont} [5], [17], [19]. SQME techniques reduce the impact of contention on WCET estimates of the monitored tasks activity by stalling contender tasks if they go beyond their quota, preventing Δ_a^{cont} of the task under analysis from being violated [1], [15], [17], [24]. SQME techniques also allow defining *contention scenarios* and derive partially time-composable WCET estimates valid under those scenarios. The basic idea is to upperbound the number of accesses of each type that potential contenders will put on the different shared resources and derive WCET estimates using those bounds. With this approach, WCET estimates are time-composable as long as the load put by contenders once the system is deployed is equal or lower to the one used to obtain these estimates.

TABLE I: Event and latency values

$L_{max}^{bus,lh}$	$L_{max}^{bus,sh}$	$L_{max}^{bus,lm}$	$L_{max}^{bus,sm}$
5	10	50	100

TABLE II: Example of the evolution of the quota assigned to τ_b. Each row corresponds to an *iteration* of the SQME solutions

Quota ($eb^{bus,xx}$)				Consumed ($ev^{bus,yy}$)				$\Delta_{b\rightarrow a}^{cont}$
xx=lh	xx=sh	xx=lm	xx=sm	yy=lh	yy=sh	yy=lm	yy=sm	
20	40	14	8	20	10	2	1	1600
16	22	8	9	6	22	5	2	900
10	15	6	4	8	15	4	3	210
4	4	1	1	3	2	0	1	75

III. Problem Statement

Existing SQME focus on a single shared resource – usually the memory as it concentrates a big fraction of the contention tasks can suffer – and a single type of request accessing that resource. However, in general, multicores comprise several shared resources $\mathcal{R}$ each of which can accept different types $\mathcal{Y}_r$ of requests. Also requests can have different access times and hence, cause different contention delays. The worst-case (longest) contention τ_b can cause on τ_a, i.e. $\Delta_{b\rightarrow a}^{cont}$, is computed as shown Equation 1.

$$\Delta_{b\rightarrow a}^{cont} = \sum_{r\in\mathcal{R}} \sum_{y\in\mathcal{Y}_r} cr_{b\rightarrow a}^{r,y} \times L_{max}^{r,y} \tag{1}$$

The contention τ_a can suffer from its contenders $c(\tau_a)$ is:

$$\Delta_a^{cont} = \sum_{\tau_x \in c(\tau_a)} \sum_{r\in\mathcal{R}} \sum_{y\in\mathcal{Y}_r} cr_{x\rightarrow a}^{r,y} \times L_{max}^{r,y} \tag{2}$$

For the single request type model, a contender task τ_b is suspended once it consumes its quota, i.e. it performs $cr_{b\rightarrow a}$ accesses. In other words, if τ_b requests are below $cr_{b\rightarrow a}$, its contention time quota is not exhausted.

When there are several types of requests, there are several combinations of event counts that lead to a quota violation and hence, event counts need to be conservative to detect any potential violation. This is better explained with an example. Let $e_i^{r,y}$ be the event monitor that counts the number of accesses of type y to resource r from core i and $eb_i^{r,y}$ the respective budget allocated to a given task. For the sake of this example, let us assume a multicore processor with the bus connecting the cores with a shared L2 cache as the only shared resource. Further assume that the L2 is partitioned and each core accesses its own private memory controller, so contention can only happen in the bus. Four types of requests can be sent to the bus: load or store (write) accesses that can hit or miss in the L2 (lh, sh, lm, sm). Each of these requests has a different maximum latency as shown in Table I. Finally, let us assume that τ_b is assigned a contention budget $\Delta_{b\rightarrow a}^{cont} = 2000$ cycles.

Iteration 1: Row 1 in Table II (quota columns) shows one potential way in which quotas on access counts can be set to prevent τ_b to cause at most $\Delta_{b\rightarrow a}^{cont}$ contention cycles on τ_a. In particular ($eb^{bus,lh}$=20, $eb^{bus,sh}$=40, $eb^{bus,lm}$=14, $eb^{bus,sm} = 8$) that for short we represent as (20,40,14,8). Once

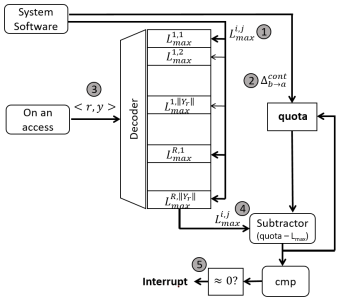

Fig. 1: Main blocks of the MCCU

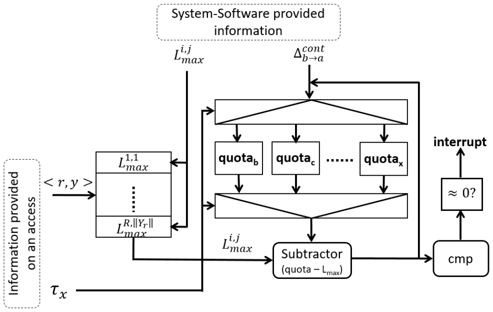

Fig. 2: MCCU for multiple tasks

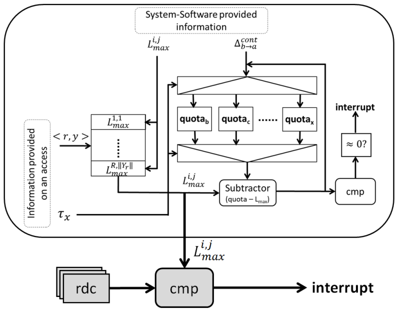

Fig. 3: Blocks to check $L_{max}^{i,j}$ is not exceeded

these values are programmed in the PMCs, τ_b is allowed to run concurrently with τ_a.

Let assume that, during its execution, τ_b makes accesses ($ev^{bus,lh}$=20,$ev^{bus,sh}$=10, $ev^{bus,lm}$=2, $ev^{bus,sm} = 1$) that we represent as $(20, 10, 2, 1)$. When τ_b makes its 20^{th} load hit access to the bus, an interrupt is raised since the $eb^{bus,lh}$ quota is exhausted. The remaining contention budget is derived as:

$$\Delta_{b\to a}^{cont}{=}2000-(20\times 5+10\times 10+2\times 50+1\times 100){=}1600$$

Iteration 2: Row 2 in Table II, under quota, shows one potential way in which quotas on access counts can be set so that once τ_b runs again, it cannot create more contention than allowed, i.e. $\Delta_{b\to a}^{cont} \leq 1600$: (16, 22, 8, 9). At the end of this second step the accesses performed by τ_b are (6,22,5,2) leaving a contention budget of $1600 - 700 = 900$ cycles.

This process repeats, generating an interrupt per iteration, until the contention budget is lower than the highest L_{max}.

In general, the need for pre-programming all $eb_i^{r,y}$ leads to cases where one counter is exhausted earlier than the others, thus raising an interrupt *despite contention budget is not yet exhausted*. This process repeats until the remaining budget is small enough not to be worth to continue with the execution. In fact, the smaller the remaining quota is, the more often interrupts are generated normally. Overall, monitoring the use of multiple shared resources in software (i.e. with SQME) just triggering interrupts only when actually exceeding contention quota is not possible. This results in significant overheads as we quantify in the Evaluation Section.

IV. MCCU

We propose the MCCU, a software-controllable low-overhead hardware unit to accurately handle several shared resources dealing with several request types, each with different access latency. We implement the MCCU as a new system-on-chip component that is attached to the on-chip bus (e.g. AMBA). This avoids introducing additional modifications to the rest of processor components (FUBs), thus drastically reducing the costs of re-design and re-verification.

A. Control logic and hardware

For each resource and request type to be tracked, the MCCU keeps $L_{max}^{r,y}$ in a register so a total of $||\mathcal{R}|| \cdot ||\mathcal{Y}_r||$ registers are needed, see Figure 1. Also one quota register per core is required to save the remaining quota cycles Δ^{cont}.

Prior to the execution of the program (e.g. at boot time), all $L_{max}^{r,y}$ are initialized ① sequentially via a single write port, see Figure 1. Since this step is done once, its overhead – in the order of dozens or hundreds of processor cycles – is negligible.

At task boundary, or software partition boundary like those in avionics ARINC 653 [2], ② the Operating System (OS) programs the MCCU with the corresponding quota, Δ^{cont}. The MCCU can be attached to the on-chip bus as a slave, e.g. AMBA Peripheral Bus (APB) slave, so that the quota counter and the registers storing the contention associated to each resource and access type can be configured and accessed using specific memory addresses, the APB addressable space.

While less frequent in CRTES, some deployments may allow task migration across cores. This requires that, whenever a task is swapped out, the OS saves its quota register in the corresponding `task struct` that the OS uses for the task. When the task is swapped back in, the OS restores the quota register. Note that $L_{max}^{r,y}$ values are intrinsic to the processor and hence, the OS can keep a single copy that can restore at boot time.

Upon each access of type y to resource r ($< r, y >$), the corresponding $L_{max}^{r,y}$ latency for that access is retrieved ③. Then, ④ the upper-bound latency of the access is subtracted from the remaining quota, which is properly updated. If the quota remaining is zero or it is close enough (e.g. it is below the latency of some access types), ⑤ an interrupt is triggered by forwarding this signal to the interrupt controller, indicating that the quota of τ_b has been exhausted.

B. Extension for Several Contenders

Interestingly, when dealing with several tasks, the table with maximum contention values per resource and access type does not need to be replicated. Instead, we only need to set a quota register per task, and use the task identifier (AMBA master signal) to retrieve the appropriate quota value and update it conveniently, see Figure 2. This approach allows setting specific quotas for each task so that, if one of them overruns its budget, the appropriate interrupt is triggered, but the other tasks remain unaffected. This approach is also suitable for scenarios with several critical tasks and contenders since the MCCU allows to set quotas for all (monitored) contending tasks and the maximum contention they could generate is independent of the particular task under analysis. Hence, the MCCU does not monitor contention for each task under analysis and contender task pair, but instead, it only monitors contention generated per task individually. For instance, let us assume a 4-core processor where τ_a is allowed to run without any quota control, τ_b – despite being critical – has a specific quota to limit the contention it may cause on τ_a (e.g. 10,000 cycles), and τ_c and τ_d are non-critical tasks.

In this context, contention quota for τ_c and τ_d must be set as the lowest contention they are allowed to produce individually on τ_a and τ_b. Also τ_b quota corresponds to the maximum contention it is allowed to cause on τ_a. Finally, since τ_a has no quota, it is programmed either with a special value so that it is ignored, or it is simply set to the maximum value possible (e.g. $2^{32}-1$) so that it cannot overrun its quota in practice. Overall the contention each task τ_x can generate ($\Delta_{x\rightarrow}$) is:

$$\Delta_{a\rightarrow} = 2^{32}-1 \qquad \Delta_{b\rightarrow} = 10,000;$$
$$\Delta_{c\rightarrow} = min(\Delta_{c\rightarrow a}, \Delta_{c\rightarrow b}); \quad \Delta_{d\rightarrow} = min(\Delta_{d\rightarrow a}, \Delta_{d\rightarrow b})$$

C. Seamless verification of $L^{i,j}_{max}$ bounds

$L^{i,j}_{max}$ is estimated empirically in real architectures, which always leaves some residual risk that estimates do not bound maximum latencies. While extensive testing helps reducing this residual risk, the MCCU allows for a seamless verification of $L^{i,j}_{max}$ bounds with a minor extension. In particular, upon each access to shared resources, the MCCU receives as input the request type and retrieves the corresponding $L^{i,j}_{max}$ bound. As shown in the bottom part of Figure 3, by simply monitoring when requests start and finish with a request duration counter, rdc, the request duration can be compared with $L^{i,j}_{max}$ and, upon an exceedance, raise an interrupt.

This allows detecting inaccuracies in the derivation of $L^{r,y}_{max}$ that might not be detected during the testing phase, which could affect the reliability of the software-estimated quotas.

In the context of AMBA, *locked* transfers allow masters to keep the ownership of the bus until the transfer completes. This enables masters to keep the bus locked indefinitely, without the arbiter being able to relinquish the grant from that master. Lock transfers are typically used to manage atomic operations such as read-modify-write, needed for synchronization. To handle locked transfers with the MCCU, we can set up a specific L_{max} register for locked transfers (or several of them if multiple types of locked transfers exist), and manage them as any other type of request. Additionally, since locked requests may be unbounded, we keep track of the duration of the in-flight request with a request duration counter (rdc), see Figure 3. At request completion time, we check whether it exceeds the corresponding $L^{r,y}_{max}$. We also check whether the locked request takes longer than *any* $L^{r,y}_{max}$ even if it is not yet finished. In both cases, an interrupt is raised, indicating that evidence has been found that the derived bounds have been violated. This is fundamental to increase confidence – reduce the residual risk in accordance with standards such as ISO 26262 in automotive.

D. Hardware Cost Analysis

Area. the MCCU requires ① a $L^{r,y}_{max}$ register for each resource and request type to be tracked. Note however, that registers are not replicated per core. Hence the number of registers needed is given by $\Delta^{cont}_{b\rightarrow a}$ and $||\mathcal{R}|| \cdot ||\mathcal{Y}_r||$. In addition, a ② 'quota' register is needed per core to keep the contention budget each core has available. Also ③ a request duration counter, rdc, per core is needed. In our reference SoC architecture, see Figure 4, the access to the bus does not implement split bus transactions so that by controlling the access to the bus we also control the access to memory. If split bus transactions are implemented, then the MCCU is also connected to the L2-to-memory interface for monitoring purposes. The L2 cache is partitioned as it is the case in many commercial architectures (e.g. NGMP [4] and ARM A9 [3]). Overall the MCCU tracks 4 bus access types: loads/stores that hit/miss in the L2. The 4 L_{max} registers (often tens of cycles) are normally encoded with just 1 byte each. Also, few bytes are needed (e.g. 4) for the quota register for each of the $Nc = 4$ cores. Thus, hardware overhead is negligible (up to 24 bytes in our reference SoC: 2×4 bytes for maximum latencies and 4×4 bytes for quota registers).

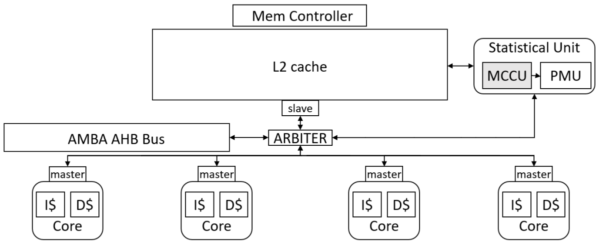

Fig. 4: Reference SoC architecture. PMU stands for performance monitoring unit that keeps the events and PMCs

Delay. As shown in Figure 3, combinational logic consists of few decoders, multiplexors and comparators, as well as a subtractor one of whose operands (L_{max}) has few bits (at most 1 byte). Hence, the area and latency of this logic is well below that of an integer Arithmetic Logic Unit, whose number, size and complexity of combinational blocks is fairly larger.

The use of a MCCU has other implications in terms of latency similar in nature to those of a PMU, but of lower magnitude. In particular, a PMU has low complexity itself, but its input consists of signals monitoring events occurring all along the processor, thus potentially taking several cycles to arrive to the PMU, which leads to event counts that can never be up-to-date and reflect event counts with some (yet limited) delay. In the case of the MCCU, events monitored are typically available as part of the AMBA AHB bus interface shown in Figure 4. Thus, connecting the MCCU to the AHB, either directly as a slave or as part of another slave (e.g. along with the PMU), makes input signals be available almost immediately when they arrive to the AHB bus. The internal processing of the MCCU, Figure 3, is simple enough to fit in one cycle or, if needed, be pipelined when the target processor frequency is high. Finally, on a quota violation there may be some delay to propagate the interrupt to the interrupt controller which, again, is limited to very few cycles in practice.

Overall, the end-to-end delay since an access starts until a potential interrupt is triggered may be in the order of 10 cycles, which could at most allow another access (e.g. the slowest one) to start, thus leading to a slight quota overrun of some tens of cycles. In the context of microcontrollers operating at several hundreds of MHz at least, such quota overrun could cost at most around 100ns, which is an insignificant impact for systems whose response time is in the order of hundreds of milliseconds (e.g. a braking system).

Scalability. For larger multicores, the $L^{r,y}_{max}$ registers do not depend on the number of cores, but on the number and type of requests. Only one quota register is needed per core. Hence, the absolute hardware cost increases negligibly. Latency wise, signal propagation may increase MCCU latencies. Also, MCCU accesses latency can increase due to access serialization to the MCCU to keep a single read/write port. This can delay by few cycles MCCU access, in the worst case, when several cores try to access the MCCU simultaneously. In both cases, the impact of the increase of few processor cycles is negligible as described in the previous paragraph.

V. Evaluation

A. *Experimental framework*

We evaluate the MCCU on a reference NGMP multicore architecture [4]. In particular, we use its implementation in a SoCLib-based [21] performance simulator with negligible performance discrepancies against the real hardware [12]: 1% on average and 3% at most. Our processor setup, see Figure 4, consists of 4 LEON4 cores including 16KB 4-way write-through L1 caches connected through an on-chip round-robin bus to a 4-way 256KB shared write-back L2 cache and a shared memory controller. The NGMP allows the L2 cache to be way-partitioned so that each core receives one way.

The MCCU tracks the 4 access types in Section III: lh, sh, lm and sm. Since the L2 is writeback, L2 misses evicting dirty data generate two memory accesses: one to write back dirty data and one to retrieve the data requested.

Following the methodology of previous works [7], [16], the latencies for the different types of request to the bus have been derived empirically, resulting in these values: $L_{max}^{bus,lh} = 10$, $L_{max}^{bus,sh} = 3$, $L_{max}^{bus,lm} = 32$, and $L_{max}^{bus,sm} = 37$.

We use the well-known MediaBench benchmark suite [14], which includes communications and multimedia functions increasingly relevant for many CRTES domains with autonomous vehicles. We only excluded `mpeg2.encode` benchmark due to issues executing it in our reference platform.

B. *Scenarios evaluated*

As explained in Section IV-B, the contention quota assigned to each (contender) task is determined by the minimum amount of contention its sibling tasks can afford.

We analyze several scenarios in which each (contender) task is allowed to cause variable contention on the task under analysis. In some scenarios, the contender task does not exceed its assigned contention quota (and hence should not be suspended). In particular, we allocate a quota 5%, 10%, ...25% higher than needed, represented in the corresponding figures as 1.05, 1.1, ... 1.25 respectively. In other scenarios, the contender task exceeds its quota, so it must be suspended. In particular it is allocated a quota smaller than the actual contention it generates according to the model (Equation 2). In particular we cover the values 0.8, 0.85, 0.9, 0.95, 1.0.

For SQME, given a maximum contention budget τ_b can generate on τ_a, $\Delta_{b\rightarrow a}^{cont}$, there are several ways in which bounds to access counts can be assigned. For instance, assuming a single resource with two request types of latencies 10 and 20 cycles, and a contention quota of 100 cycles, the possible ways in which bounds to access counts can be assigned are: $(10, 0), (8, 1), (6, 2), (4, 3), (2, 4), (0, 5)$. Determining the best access bounds a priori is challenging since the number of accesses of each type of a program can depend on factors such as input data and vary throughout program execution. Hence, any a-priori choice, which we refer to as request bound breakdown scenarios (*rbbs*), is arbitrary.

We have explored two *rbbs*. In both cases, in the first iteration, we distribute contention quota homogeneously across request types. Hence, each request type y_r is given a *FairShare* of the overall quota:

$FairShare = (\Delta_{b\rightarrow a}^{cont})/(||\mathcal{R}|| \cdot ||\mathcal{Y}_{\triangledown}||)$.

For each request type $eb^{r,y}$, requests are allowed as defined next: $eb^{r,y} = \lfloor FairShare/L_{max}^{r,y} \rfloor$. Whenever the quota for a given request type is exhausted, in the following iterations we either follow the same approach based on a fair quota share ($rbbs_{fair-fair}$), or we assign the quota proportionally to the consumed quota across accesses so far ($rbbs_{fair-prop}$).

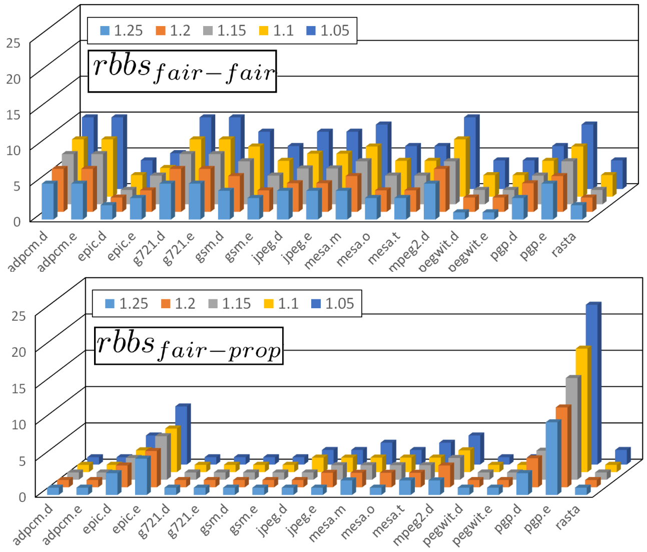

Fig. 5: Number of interrupts generated by SW solutions.

For instance, let us assume that the quota is 1,000 cycles and two request types with $L_{max}^{r,y}$ 10 and 20 cycles respectively. In this case, we allocate 500 cycles to each request type in the first iteration, thus granting 50 and 25 accesses respectively. Let us now assume that the task performs 50 and 5 accesses respectively, thus exhausting the quota for the first access type. The remaining quota would be 400 cycles. Under $rbbs_{fair-fair}$, we would allocate 200 cycles to each request type, so 20 and 10 accesses. Under $rbbs_{fair-prop}$, instead, we would take into account that each access type has used 500 and 100 cycles respectively, so we would split the remaining 400 cycles keeping the same ratio, so 333 and 67, thus granting 33 and 6 accesses of each type respectively.

C. *Sufficient contention quota*

1) SQME results: Figure 5 shows the number of interrupts raised by each task under the different scenarios in which tasks have sufficient budget. It is worth mentioning that the latency of an interrupt is the time between the start of an Interrupt Request (IRQ) and the completion of the respective Interrupt Service Routine (ISR). The direct cost of interrupts together with their impact on processor state (e.g. cache state) impact execution time (and WCET) in non-obvious ways. Moreover, just predicting a priori the number of interrupts to account for them in the WCET is challenging. Therefore, in addition to their actual cost, unnecessary interrupts caused by SQME challenge the derivation of reliable and tight WCET estimates.

In Figure 5, the z-axis shows the quotas that range from 1.05x to 1.25x of the actual quota each benchmark (on the x-axis) would need in practice. As shown, the number of unnecessarily-generated interrupts is high for both $rbbs_{fair-fair}$ (top plot) and $rbbs_{fair-prop}$ (bottom plot) software approaches and increases as the contention quota decreases down to 1.05. For $rbbs_{fair-fair}$, interrupts are quite stable around 5, while for $rbbs_{fair-prop}$, on average, the number of interrupts decreases, and in many cases is just 1. For

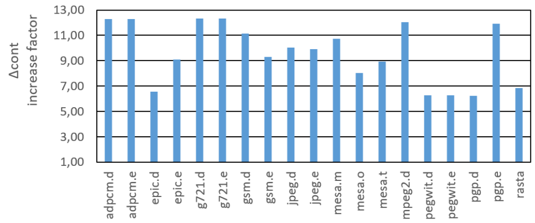

Fig. 6: Increase in $\Delta^{cont}_{b\to a}$ due to the simple support for the single-request approach w.r.t. MCCU.

some benchmarks such as `epic.d` and `pgp.e`, they increase significantly due to the varying behavior of the benchmarks over time. Either the case, software-only solutions (SQME) need to allocate quota statically to request types, thus causing unnecessary interrupts due to their inability to allocate quota dynamically to the request types that occur in practice.

Resorting back to the single-request solution. A simple approach overcome the problem of distributing the quota across request types, consists of assuming a minimal hardware support that adds the desired events in a given counter. In our case, $ev^{comb} = ev^{lh} + ev^{sh} + ev^{lm} + ev^{sm}$. That is, all requests are assumed to be of the same type. In the CRTES domain, this necessarily means to assume that all requests are from the worst type, so with the highest latency, which requires reformulating Equation 2 as follows.

$$\Delta^{cont}_{b\to a} = \max_{r\in\mathcal{R},y\in\mathcal{Y}_r} (L^{r,y}_{max}) \times \left(\sum_{r\in\mathcal{R}} \sum_{y\in\mathcal{Y}_r} cr^{r,y}_{b\to a} \right) \qquad (3)$$

While this solution would certainly avoid generating unnecessary interrupts, assuming that all requests are from the worst type implies that quota is consumed – pessimistically – much faster, leading to a single (but much earlier) interrupt.

In order to show the impact of this approach in our reference architecture, we have measured in Figure 6 the increment in contention delay ($\Delta^{cont}_{b\to a}$) of Equation 3 with respect to Equation 1, the latter of which is captured by the hardware model in the MCCU.

The observed increase for MediaBench in our reference architecture is $9.6x$ on average and as high as $12.3x$. This indicates that quota is consumed at a $9.6x$ higher rate with this approach, which is simply an unaffordable overhead that makes this solution not attractive either. In fact, this would require that the task under analysis could afford $9.6x$ higher contention than with the MCCU.

2) MCCU: For the MCCU, zero unnecessary interrupts are generated. Since quota suffices, the contender task will always complete its execution before exhausting its quota.

VI. Conclusions

We present the MCCU, a low-overhead hardware component that allows allocating contention quotas to tasks for shared hardware resources and monitoring whether those quotas are exhausted. The MCCU raises one interrupt only when strictly needed, hence avoiding the limitations of software-only solutions that allocate individual quotas per resource and access type resulting in frequent unnecessary interrupts. Moreover, the MCCU allows assessing seamlessly whether the maximum latency per request type estimated is exceeded, thus detecting potential issues in the WCET estimation process. Finally, the hardware overhead of the MCCU is quite limited and does not require re-designing or re-verifying existing FUBs, which would challenge its adoption due to cost reasons. Our MCCU can be applied into other models even though that will be addressed in future work.

VII. Acknowledgments

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773) and the HiPEAC Network of Excellence. Carles Hernández is jointly funded by the MINECO and FEDER funds through grant TIN2014-60404-JIN. Jaume Abella has been partially supported by the MINECO under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

References

[1] A. Agrawal et al. Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study. In *ECRTS*, 2017.

[2] ARINC Inc. *ARINC Specification 653: Avionics Application Software Standard Standard Interface, Part 1 and 4, Subset Services*, June 2012.

[3] ARM Ltd. *ARM Cortex-A9 Technical Reference Manual r4p1*, 2016.

[4] Cobham Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and Users Manual*, 2011.

[5] D. Dasari et al. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *IEEE TrustCom*, 2011.

[6] E. Díaz et al. MC2: Multicore and cache analysis via deterministic and probabilistic jitter bounding. In *Ada-Europe*, 2017.

[7] M. Fernández et al. Assessing the suitability of the ngmp multi-core processor in the space domain. In *EMSOFT*, 2012.

[8] E. Francis. Autonomous cars: no longer just science fiction. *Automotive Industries*, 193, 2014.

[9] A. Hansson et al. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Design Autom. Electr. Syst.*, 14:2:1–2:24, 2009.

[10] C. Hernández et al. Design and implementation of a time predictable processor: Evaluation with a space case study. In *ECRTS*, 2017.

[11] J. Jalle et al. Bounding Resource Contention Interference in the Next-Generation Microprocessor (NGMP). In *ERTS*, 2015.

[12] J. Jalle et al. Validating a timing simulator for the NGMP multicore processor. In *DASIA*, 2016.

[13] P. Kumar et al. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS*, 2016.

[14] C. Lee et al. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO*, 1997.

[15] R. Mancuso et al. Wcet(m) estimation in multi-core systems using single core equivalence. In *ECRTS*, 2015.

[16] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *EDCC*. IEEE Computer Society, 2012.

[17] J. Nowotsch et al. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.

[18] M. Paolieri et al. Hardware support for wcet analysis of hard real-time multicore systems. In *ISCA*, 2009.

[19] R. Pellizzoni et al. Worst case delay analysis for memory interference in multicore systems. In *DATE*, 2010.

[20] M. Schoeberl and A. Rocha. T-crest: A time-predictable multi-core platform for aerospace applications. In *DASIA*, 2014.

[21] SoCLib. -, 2003-2012. http://www.soclib.fr/trac/dev.

[22] K. Suleman. Intel paves the road for bmw's inext autonomous cars in 2021. 2017.

[23] T. Ungerer et al. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010.

[24] H. Yun et al. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*, 2013.

[25] M. Zimmer et al. Flexpret: A processor platform for mixed-criticality systems. In *RTAS*, pages 101–110, 2014.